

About this Book

The super-fast evolution of the JDK between versions 8 and 12 has increased the learning curve of modern Java, therefore has increased the time needed for placing developers in the Plateau of Productivity. Its new features and concepts can be adopted to solve a variety of modern-day problems. This book enables you to adopt an objective approach to common problems by explaining the correct practices and decisions with respect to complexity, performance, readability, and more.

Java Coding Problems will help you complete your daily tasks and meet deadlines. You can count on the 300+ applications containing 1,000+ examples in this book to cover the common and fundamental areas of interest: strings, numbers, arrays, collections, data structures, date and time, immutability, type inference, Optional, Java I/O, Java Reflection, functional programming, concurrency and the HTTP Client API. Put your skills on steroids with problems that have been carefully crafted to highlight and cover the core knowledge that is accessed in daily work. In other words (no matter if your task is easy, medium or complex) having this knowledge under your tool belt is a must, not an option.

By the end of this book, you will have gained a strong understanding of Java concepts and have the confidence to develop and choose the right solutions to your problems.

Publication date:

septiembre 2019

Publisher

Packt

Pages

816

ISBN

9781789801415

Repository book:

<https://github.com/packtpublishing/java-coding-problems>

Preface

The super-fast evolution of the JDK between versions 8 and 12 has increased the learning curve of modern Java, therefore has increased the time needed for placing developers in the Plateau of Productivity. Its new features and concepts can be adopted to solve a variety of modern-day problems. This book enables you to adopt an objective approach to common problems by explaining the correct practices and decisions with respect to complexity, performance, readability, and more.

*Java Coding Problems* will help you complete your daily tasks and meet deadlines. You can count on the 300+ applications containing 1,000+ examples in this book to cover the common and fundamental areas of interest: strings, numbers, arrays, collections, data structures, date and time, immutability, type inference, Optional, Java I/O, Java Reflection, functional programming, concurrency and the HTTP Client API. Put your skills on steroids with problems that have been carefully crafted to highlight and cover the core knowledge that is accessed in daily work. In other words (no matter if your task is easy, medium or complex) having this knowledge under your tool belt is a must, not an option.

By the end of this book, you will have gained a strong understanding of Java concepts and have the confidence to develop and choose the right solutions to your problems.

Who this book is for

*Java Coding Problems* is especially useful for beginners and intermediate Java developers. However, the problems looked at here are encountered in the daily work of any Java developer.

The required technical background is quite thin. Mainly, you should be a Java fan and have good skills and intuition in following a piece of Java code.

What this book covers

[Chapter 1](https://subscription.packtpub.com/book/programming/9781789801415/1), *Strings, Numbers, and Math*, includes 39 problems that involve strings, numbers, and mathematical operations. The chapter starts with a bunch of classical problems for strings such as counting duplicates, reversing a string, and removing white spaces. The chapter continues with problems dedicated to numbers and mathematical operations such as summing two large numbers, operation overflow, comparing two unsigned numbers, computing the floor of a division and a modulus, and much more. Each problem is passed through several solutions, including Java 8 functional style. Moreover, the chapter covers problems that futures added in JDK 9, 10, 11, and 12.

[Chapter 2](https://subscription.packtpub.com/book/programming/9781789801415/2), *Objects, Immutability, and Switch Expressions*, includes 18 problems that involve objects, immutability, and switch expressions. The chapter starts with several problems about dealing with null references. It continues with problems regarding checking indexes, equals() and hashCode(), and immutability (for example, writing immutable classes and passing/returning mutable objects from immutable classes). The last part of the chapter deals with cloning objects and JDK 12 switch expressions.

[Chapter 3](https://subscription.packtpub.com/book/programming/9781789801415/3), *Working with Date and Time*, includes 20 problems that involve date and time. These problems are meant to cover a wide range of topics (converting, formatting, adding, subtracting, defining periods/durations, computing, and so on) via Date, Calendar, LocalDate, LocalTime, LocalDateTime, ZoneDateTime, OffsetDateTime, OffsetTime, Instant, and so on. By the end of this chapter, you will have no problems shaping date and time to conform to your application's needs.

[Chapter 4](https://subscription.packtpub.com/book/programming/9781789801415/4), *Type Inference*, includes 21 problems that involve JEP 286, Java **Local Variable Type Inference** (**LVTI**), or the var type. These problems have been carefully crafted to reveal the best practices and common mistakes involved in using var. By the end of this chapter, you will have everything you need to know about var to push it in production.

[Chapter 5](https://subscription.packtpub.com/book/programming/9781789801415/5), *Arrays, Collections, and Data Structures*, includes 30 problems that involve arrays, collections, and several data structures. The aim is to provide solutions to a category of problems encountered in a wide range of applications, such as sorting, finding, comparing, ordering, reversing, filling, merging, copying, replacing, and so on. The provided solutions are implemented in Java 8-12 and they can be used as the base for solving other related problems as well. By the end of this chapter, you will have a solid base of knowledge that's useful for solving a lot of problems that involve arrays, collections, and data structures.

[Chapter 6](https://subscription.packtpub.com/book/programming/9781789801415/6), *Java I/O Paths, Files, Buffers, Scanning, and Formatting*, includes 20 problems that involve Java I/O for files. From manipulating, walking, and watching paths to streaming files and efficient ways for reading/writing text and binary files, we will cover problems that are a must in the arsenal of any Java developer. With the skills gained from this chapter, you will be able to tackle most of the common problems that involve Java I/O files.

[Chapter 7](https://subscription.packtpub.com/book/programming/9781789801415/7), *Java Reflection Classes, Interfaces, Constructors, Methods, and Fields*, includes 17 problems that involve the Java Reflection API. From classical topics, such as inspecting and instantiating Java artifacts (for example, modules, packages, classes, interfaces, super-classes, constructors, methods, annotations, arrays, and so on), to synthetic and bridge constructs or nest-based access control (JDK 11), this chapter provides solid coverage of the Java Reflection API.

[Chapter 8](https://subscription.packtpub.com/book/programming/9781789801415/8), *Functional Style Programming – Fundamentals and Design Patterns*, includes 11 problems that involve Java functional programming. The chapter starts with a problem designed to acquaint you completely with functional interfaces. It continues with a suite of design patterns from GoF interpreted in Java functional style.

[Chapter 9](https://subscription.packtpub.com/book/programming/9781789801415/9), *Functional Style Programming – Deep Dive*, includes 22 problems that involve Java functional programming. Here, we focus on several problems that involve classical operations encountered in streams (for example, filters, and maps), and we discuss infinite streams, null-safe streams, and default methods. A comprehensive list of problems covers grouping, partitioning, and collectors, including the JDK 12 teeing() collector and the matter of writing a custom collector. In addition, takeWhile(), dropWhile(), composing functions, predicates and comparators, testing and debugging lambdas, and other cool topics are discussed as well.

[Chapter 10](https://subscription.packtpub.com/book/programming/9781789801415/10), *Concurrency – Thread Pools, Callables, and Synchronizers*, includes 14 problems that involve Java concurrency. This chapter starts with several fundamental problems about the thread life cycle and object-/class-level locking. It continues with a bunch of problems about thread pools in Java, including JDK 8 work-stealing thread pools. Afterward, we have problems dedicated to Callable and Future. Next, we dedicate several problems to Java synchronizers (for example, barrier, semaphore, and exchanger). By the end of this chapter, you should be familiar with the main coordinates of Java concurrency and be ready to continue with a set of advanced problems.

[Chapter 11](https://subscription.packtpub.com/book/programming/9781789801415/11), *Concurrency – Deep Dive*, includes 13 problems that involve Java concurrency. This chapter covers problems about fork/join frameworks, CompletableFuture, ReentrantLock, ReentrantReadWriteLock, StampedLock, atomic variables, task cancelation, interruptible methods, thread-local locks, and deadlocks. Completing this chapter will guarantee the achievement of the considerable amount of concurrency knowledge needed by any Java developer.

[Chapter 12](https://subscription.packtpub.com/book/programming/9781789801415/12), *Optional*, includes 24 problems meant to draw several rules for working with Optional. The problems and solutions presented in this section are based on the Brian Goetz' (Java's language architect) definition—*Optional* *is intended to provide a limited mechanism for library method return types where there needed to be a clear way to represent no result, and using null for such was overwhelmingly likely to cause errors.* But where there are rules, there are exceptions as well. Therefore, do not conclude that the rules (or practices) presented here should be followed (or avoided) at all costs. Like always, the solution depends on the problem.

[Chapter 13](https://subscription.packtpub.com/book/programming/9781789801415/13), *HTTP Client and WebSocket APIs*, includes 20 problems meant to cover the HTTP Client and WebSocket APIs. Remember HttpUrlConnection? Well, JDK 11 comes with the HTTP Client API as a reinvention of HttpUrlConnection. The HTTP Client API is easy to use and supports HTTP/2 (default) and HTTP/1.1. For backward compatibility, the HTTP Client API will automatically downgrade from HTTP/2 to HTTP 1.1 when the server doesn't support HTTP/2. Moreover, the HTTP Client API supports synchronous and asynchronous programming models and relies on streams to transfer data (reactive streams). It also supports the WebSocket protocol used in real-time web applications to provide client-server communication with low message overhead.

To get the most out of this book

You should have fundamental knowledge about the Java language. You should install the following:

* An IDE (recommended, but not a must, is Apache NetBeans 11.x: <https://netbeans.apache.org/>)
* JDK 12 and Maven 3.3.x
* Additional third-party libraries will need to be installed at the right moment (nothing special)

Download the example code files

You can download the example code files for this book from your account at [www.packt.com](http://www.packt.com/). If you purchased this book elsewhere, you can visit [www.packtpub.com/support](https://www.packtpub.com/support) and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at [www.packt.com](http://www.packt.com/).
2. Select the Support tab.
3. Click on Code Downloads.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

* WinRAR/7-Zip for Windows
* Zipeg/iZip/UnRarX for Mac
* 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Java-Coding-Problems>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at [**https://github.com/PacktPublishing/**](https://github.com/PacktPublishing/). Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://static.packt-cdn.com/downloads/9781789801415_ColorImages.pdf>.

Code in action

To see the code being executed please visit the following link: <http://bit.ly/2kSgFKf>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "If the current character exists in the Map instance, then simply increase its occurrences by 1 with (character, occurrences+1)."

A block of code is set as follows:

public Map<Character, Integer> countDuplicateCharacters(String str) {  
  
 Map<Character, Integer> result = new HashMap<>();  
  
 // or use for(char ch: str.toCharArray()) { ... }  
 for (int i = 0; i<str.length(); i++) {  
 char ch = str.charAt(i);   
  
 result.compute(ch, (k, v) -> (v == null) ? 1 : ++v);  
 }  
  
 return result;  
}Copy

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

for (int i = 0; i < str.length(); i++) **{**  
 **int cp = str.codePointAt(i);**  
 **String ch = String.valueOf(Character.toChars(cp));**  
 **if(Character.charCount(cp) == 2) { // 2 means a surrogate pair**  
 **i++;**  
 **}**  
}Copy

Any command-line input or output is written as follows:

**$ mkdir css**  
**$ cd css**Copy

**Bold**: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "In Java, the logical **AND** operator is represented as &&, the logical **OR** operator is represented as ||, and the logical **XOR** operator is represented as ^."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

**General feedback**: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

**Errata**: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](https://www.packtpub.com/support/errata), selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy**: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

**If you are interested in becoming an author**: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com/).

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit [packt.com](http://www.packt.com/).

Strings, Numbers, and Math

This chapter includes 39 problems that involve strings, numbers, and mathematical operations. We will start by looking at a bunch of classical problems for strings such as counting duplicates, reversing a string, and removing white spaces. Then, we will look at problems dedicated to numbers and mathematical operations such as summing two large numbers and operation overflow, comparing two unsigned numbers, and computing the floor of a division and modulus. Each problem is passed through several solutions, including Java 8's functional style. Moreover, we will be covering problems that concern JDK 9, 10, 11, and 12.

By the end of this chapter, you will know how to use a bunch of techniques so that you can manipulate strings and apply, adapt, and adjust them to many other problems. You will also know how to solve mathematical corner cases that may lead to weird and unpredictable results.

Problems

Use the following problems to test your string manipulation and mathematical corner case programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Counting duplicate characters**: Write a program that counts duplicate characters from a given string.
2. **Finding the first non-repeated character**: Write a program that returns the first non-repeated character from a given string.
3. **Reversing letters and words**: Write a program that reverses the letters of each word and a program that reverses the letters of each word and the words themselves.
4. **Checking whether a string contains only digits**: Write a program that checks whether the given string contains only digits.
5. **Counting vowels and consonants**: Write a program that counts the number of vowels and consonants in a given string. Do this for the English language, which has five vowels (a, e, i, o, and u).
6. **Counting occurrences of a certain character**: Write a program that counts the occurrences of a certain character in a given string.
7. **Converting String into int, long, float, or double**: Write a program that converts the given String object (representing a number) into int, long, float, or double.
8. **Removing white spaces from a string**: Write a program that removes all white spaces from the given string.
9. **Joining multiple strings with a delimiter**: Write a program that joins the given strings by the given delimiter.
10. **Generating all permutations**: Write a program that generates all of the permutations of a given string.
11. **Checking whether a string is a palindrome**: Write a program that determines whether the given string is a palindrome or not.
12. **Removing duplicate characters**: Write a program that removes the duplicate characters from the given string.
13. **Removing given characters**: Write a program that removes the given character from the given string.
14. **Finding the character with the most appearances**: Write a program that finds the character with the most appearances in the given string.
15. **Sorting an array of strings by length**: Write a program that sorts by the length of the given array of strings.
16. **Checking that a string contains a substring**: Write a program that checks whether the given string contains the given substring.
17. **Counting substring occurrences a string**: Write a program that counts the occurrences of a given string in another given string.
18. **Checking whether two strings are anagrams**: Write a program that checks whether two strings are anagrams. Consider that an anagram of a string is a permutation of this string by ignoring capitalization and white spaces.
19. **Declaring multiline strings (text blocks)**: Write a program that declares multiline strings or text blocks.
20. **Concatenating the same string *n* times**: Write a program that concatenates the same string a given number of times.
21. **Removing leading and trailing spaces**: Write a program that removes the leading and trailing spaces of the given string.
22. **Finding the longest common prefix**: Write a program that finds the longest common prefix of given strings.
23. **Applying indentation**: Write several snippets of code to apply indentation to the given text.
24. **Transforming strings**: Write several snippets of code to transform a string into another string.
25. **Computing the minimum and maximum of two numbers**: Write a program that returns the minimum and maximum of two numbers.
26. **Summing two large int/long numbers and operation overflow**: Write a program that sums two large int/long numbers and throws an arithmetic exception in the case of an operation overflow.
27. **String as an unsigned number in the radix**: Write a program that parses the given string into an unsigned number (int or long) in the given radix.
28. **Converting into a number by an unsigned conversion**: Write a program that converts a given int number into long by an unsigned conversion.
29. **Comparing two unsigned numbers**: Write a program that compares the given two numbers as unsigned.
30. **Division and modulo of unsigned values**: Write a program that computes the division and modulo of the given unsigned value.
31. **double/float is a finite floating-point value**: Write a program that determines whether the given double/float value is a finite floating-point value.
32. **Applying logical AND/OR/XOR to two boolean expressions**: Write a program that applies the logical AND/OR/XOR to two boolean expressions.
33. **Converting** BigInteger**into a primitive type**: Write a program that extracts the primitive type value from the given BigInteger.
34. **Converting long into int**: Write a program that converts long into int.
35. **Computing the floor of a division and modulus**: Write a program that computes the floor division and the floor modulus of the given dividend (*x*) and divisor (*y*).
36. **Next floating-point value**: Write a program that returns the next floating-point adjacent to the given float/double value in the direction of positive and negative infinity.
37. **Multiplying two large int/long values and operation overflow**: Write a program that multiplies two large int/long values and throws an arithmetic exception in the case of operation overflow.
38. **Fused Multiply Add** (**FMA**): Write a program that takes three float/double values (*a*, *b*, *c*) and computes *a \* b + c* in an efficient way.
39. **Compact number formatting**: Write a program that formats the number 1,000,000 to 1M (US locale) and to 1 mln (Italian locale). In addition, parse 1M and 1 mln from a string into a number.

Solutions

The following sections describe solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations shown here only include the most interesting and important details needed to solve the problems. You can download the example solutions to see additional details and experiment with the programs from <https://github.com/PacktPublishing/Java-Coding-Problems>.

1. Counting duplicate characters

The solution to counting the characters in a string (including special characters such as #, $, and %) implies taking each character and comparing them with the rest. During the comparison, the counting state is maintained via a numeric counter that's increased by one each time the current character is found.

There are two solutions to this problem.

The first solution iterates the string characters and uses Map to store the characters as keys and the number of occurrences as values. If the current character was never added to Map, then add it as (character, 1). If the current character exists in Map, then simply increase its occurrences by 1, for example, (character, occurrences+1). This is shown in the following code:

public Map<Character, Integer> countDuplicateCharacters(String str) {  
  
 Map<Character, Integer> result = new HashMap<>();  
  
 // or use for(char ch: str.toCharArray()) { ... }  
 for (int i = 0; i<str.length(); i++) {  
 char ch = str.charAt(i);   
  
 result.compute(ch, (k, v) -> (v == null) ? 1 : ++v);  
 }  
  
 return result;  
}Copy

Another solution relies on Java 8's stream feature. This solution has three steps. The first two steps are meant to transform the given string into Stream<Character>, while the last step is responsible for grouping and counting the characters. Here are the steps:

1. Call the String.chars() method on the original string. This will return IntStream. This IntStream contains an integer representation of the characters from the given string.
2. Transform IntStream into a stream of characters via the mapToObj() method (convert the integer representation into the human-friendly character form).
3. Finally, group the characters (Collectors.groupingBy()) and count them (Collectors.counting()).

The following snippet of code glues these three steps into a single method:

public Map<Character, Long> countDuplicateCharacters(String str) {  
  
 Map<Character, Long> result = str.chars()  
 .mapToObj(c -> (char) c)  
 .collect(Collectors.groupingBy(c -> c, Collectors.counting()));  
  
 return result;  
}Copy

What about Unicode characters?

We are pretty familiar with ASCII characters. We have unprintable control codes between 0-31, printable characters between 32-127, and extended ASCII codes between 128-255. But what about Unicode characters? Consider this section for each problem that requires that we manipulate Unicode characters.

So, in a nutshell, early Unicode versions contained characters with values less than 65,535 (0xFFFF). Java represents these characters using the 16-bit char data type. Calling charAt(i) works as expected as long as *i* doesn't exceed 65,535. But over time, Unicode has added more characters and the maximum value has reached 1,114,111 (0x10FFFF). These characters don't fit into 16 bits, and so 32-bit values (known as *code points*) were considered for the UTF-32 encoding scheme.

Unfortunately, Java doesn't support UTF-32! Nevertheless, Unicode has come up with a solution for still using 16 bits to represent these characters. This solution implies the following:

* 16-bit *high surrogates*: 1,024 values (U+D800 to U+DBFF)
* 16-bit *low surrogates*: 1,024 values (U+DC00 to U+DFFF)

Now, a high surrogate followed by a low surrogate defines what is known as a **surrogate pair**. Surrogate pairs are used to represent values between 65,536 (0x10000) and 1,114,111 (0x10FFFF). So, certain characters, known as Unicode supplementary characters, are represented as Unicode surrogate pairs (a one-character (symbol) fits in the space of a pair of characters) that are merged into a single code point. Java takes advantage of this representation and exposes it via a suite of methods such as codePointAt(), codePoints(), codePointCount(), and offsetByCodePoints() (take a look at the Java documentation for details). Calling codePointAt() instead of charAt(), codePoints() instead of chars(), and so on helps us to write solutions that cover ASCII and Unicode characters as well.

For example, the well-known two hearts symbol is a Unicode surrogate pair that can be represented as a char[] containing two values: \uD83D and \uDC95. The code point of this symbol is 128149. To obtain a String object from this code point, call String str = String.valueOf(Character.toChars(128149)). Counting the code points in str can be done by calling str.codePointCount(0, str.length()), which returns 1 even if the str length is 2. Calling str.codePointAt(0) returns 128149 and calling str.codePointAt(1) returns 56469. Calling Character.toChars(128149) returns 2 since two characters are needed to represent this code point being a Unicode surrogate pair. For ASCII and Unicode 16- bit characters, it will return 1.

So, if we try to rewrite the first solution (that iterates the string characters and uses Map to store the characters as keys and the number of occurrences as values) to support ASCII and Unicode (including surrogate pairs), we obtain the following code:

public static Map<String, Integer>   
 countDuplicateCharacters(String str) {  
   
 Map<String, Integer> result = new HashMap<>();  
  
 for (int i = 0; i < str.length(); i++) {  
 **int cp = str.codePointAt(i);**  
 **String ch = String.valueOf(Character.toChars(cp));**  
 **if(Character.charCount(cp) == 2) { // 2 means a surrogate pair**  
 **i++;**  
 **}**  
  
 result.compute(ch, (k, v) -> (v == null) ? 1 : ++v);  
 }  
  
 return result;  
}Copy

The highlighted code can be written as follows, as well:

String ch = String.valueOf(Character.toChars(str.codePointAt(i)));  
if (i < str.length() - 1 && str.codePointCount(i, i + 2) == 1) {  
 i++;  
}Copy

Finally, trying to rewrite the Java 8 functional style solution to cover Unicode surrogate pairs can be done as follows:

public static Map<String, Long> countDuplicateCharacters(String str) {   
  
 Map<String, Long> result = str.codePoints()  
 .mapToObj(c -> String.valueOf(Character.toChars(c)))  
 .collect(Collectors.groupingBy(c -> c, Collectors.counting()));  
  
 return result;  
}Copy

For third-party library support, please consider Guava: Multiset<String>.

Some of the following problems will provide solutions that cover ASCII, 16-bit Unicode, and Unicode surrogate pairs as well. They have been chosen arbitrarily, and so, by relying on these solutions, you can easily write solutions for problems that don't provide such a solution.

2. Finding the first non-repeated character

There are different solutions to finding the first non-repeated character in a string. Mainly, the problem can be solved in a single traversal of the string or in more complete/partial traversals.

In the single traversal approach, we populate an array that's meant to store the indexes of all of the characters that appear exactly once in the string. With this array, simply return the smallest index containing a non-repeated character:

private static final int EXTENDED\_ASCII\_CODES = 256;  
...  
public char firstNonRepeatedCharacter(String str) {  
  
 int[] flags = new int[EXTENDED\_ASCII\_CODES];  
  
 for (int i = 0; i < flags.length; i++) {  
 flags[i] = -1;  
 }  
  
 for (int i = 0; i < str.length(); i++) {  
 char ch = str.charAt(i);  
 if (flags[ch] == -1) {  
 flags[ch] = i;  
 } else {  
 flags[ch] = -2;  
 }  
 }  
  
 int position = Integer.MAX\_VALUE;  
  
 for (int i = 0; i < EXTENDED\_ASCII\_CODES; i++) {  
 if (flags[i] >= 0) {  
 position = Math.min(position, flags[i]);  
 }  
 }  
  
 return position == Integer.MAX\_VALUE ?  
 Character.MIN\_VALUE : str.charAt(position);  
}Copy

This solution assumes that every character from the string is part of the extended ASCII table (256 codes). Having codes greater than 256 requires us to increase the size of the array accordingly (<http://www.alansofficespace.com/unicode/unicd99.htm>). The solution will work as long as the array size is not extended beyond the largest value of the char type, which is Character.MAX\_VALUE, that is, 65,535. On the other hand, Character.MAX\_CODE\_POINT returns the maximum value of a Unicode code point, 1,114,111. To cover this range, we need another implementation based on codePointAt() and codePoints().

Thanks to the single traversal approach, this is pretty fast. Another solution consists of looping the string for each character and counting the number of occurrences. Every second occurrence (duplicate) simply breaks the loop, jumps to the next character, and repeats the algorithm. If the end of the string is reached, then it returns the current character as the first non-repeatable character. This solution is available in the code bundled with this book.

Another solution that's presented here relies on LinkedHashMap. This Java map is an *insertion-order* map (it maintains the order in which the keys were inserted into the map) and is very convenient for this solution. LinkedHashMap is populated with characters as keys and the number of occurrences as values. Once LinkedHashMap is complete, it will return the first key that has a value equal to 1. Thanks to the *insertion-order* feature, this is the first non-repeatable character in the string:

public char firstNonRepeatedCharacter(String str) {  
  
 Map<Character, Integer> chars = new LinkedHashMap<>();  
  
 // or use for(char ch: str.toCharArray()) { ... }  
 for (int i = 0; i < str.length(); i++) {  
 char ch = str.charAt(i);  
  
 chars.compute(ch, (k, v) -> (v == null) ? 1 : ++v);  
 }  
  
 for (Map.Entry<Character, Integer> entry: chars.entrySet()) {  
 if (entry.getValue() == 1) {  
 return entry.getKey();  
 }  
 }  
  
 return Character.MIN\_VALUE;  
}Copy

In the code bundled with this book, the aforementioned solution has been written in Java 8 functional style. Moreover, the functional style solution for supporting ASCII, 16-bit Unicode, and Unicode surrogate pairs is as follows:

public static String firstNonRepeatedCharacter(String str) {  
   
 Map<Integer, Long> chs = str.codePoints()  
 .mapToObj(cp -> cp)  
 .collect(Collectors.groupingBy(Function.identity(),  
 LinkedHashMap::new, Collectors.counting()));  
  
 int cp = chs.entrySet().stream()  
 .filter(e -> e.getValue() == 1L)  
 .findFirst()  
 .map(Map.Entry::getKey)  
 .orElse(Integer.valueOf(Character.MIN\_VALUE));  
  
 return String.valueOf(Character.toChars(cp));  
}Copy

To understand this code in more detail, please consider the *What about Unicode characters?* subsection of the *Counting duplicate characters*section.

3. Reversing letters and words

First, let's reverse only the letters of each word. The solution to this problem can exploit the StringBuilder class. The first step consists of splitting the string into an array of words using a white space as the delimiter (Spring.split(" ")). Furthermore, we reverse each word using the corresponding ASCII codes and append the result to StringBuilder. First, we split the given string by space. Then, we loop the obtained array of words and reverse each word by fetching each character via charAt() in reverse order:

private static final String WHITESPACE = " ";  
...  
public String reverseWords(String str) {  
  
 String[] words = str.split(WHITESPACE);  
 StringBuilder reversedString = new StringBuilder();  
  
 for (String word: words) {  
 StringBuilder reverseWord = new StringBuilder();  
  
 for (int i = word.length() - 1; i >= 0; i--) {  
 reverseWord.append(word.charAt(i));  
 }  
  
 reversedString.append(reverseWord).append(WHITESPACE);  
 }  
  
 return reversedString.toString();  
}Copy

Obtaining the same result in Java 8 functional style can be done as follows:

private static final Pattern PATTERN = Pattern.compile(" +");  
...  
public static String reverseWords(String str) {  
  
 return PATTERN.splitAsStream(str)  
 .map(w -> new StringBuilder(w).reverse())  
 .collect(Collectors.joining(" "));  
}Copy

Notice that the preceding two methods return a string containing the letters of each word reversed, but the words themselves are in the same initial order. Now, let's consider another method that reverses the letters of each word and the words themselves. Thanks to the built-in StringBuilder.reverse() method, this is very easy to accomplish:

public String reverse(String str) {  
  
 return new StringBuilder(str).reverse().toString();  
}Copy

For third-party library support, please consider the Apache Commons Lang, StringUtils.reverse().

4. Checking whether a string contains only digits

The solution to this problem relies on the Character.isDigit() or String.matches() method.

The solution relying on Character.isDigit() is pretty simple and fast—loop the string characters and break the loop if this method returns false:

public static boolean containsOnlyDigits(String str) {  
  
 for (int i = 0; i < str.length(); i++) {  
 if (!Character.isDigit(str.charAt(i))) {  
 return false;  
 }  
 }  
  
 return true;  
}Copy

In Java 8 functional style, the preceding code can be rewritten using anyMatch():

public static boolean containsOnlyDigits(String str) {  
  
 return !str.chars()  
 .anyMatch(n -> !Character.isDigit(n));  
}Copy

Another solution relies on String.matches(). This method returns a boolean value indicating whether or not this string matches the given regular expression:

public static boolean containsOnlyDigits(String str) {  
  
 return str.matches("[0-9]+");  
}Copy

Notice that Java 8 functional style and regular expression-based solutions are usually slow, so if speed is a requirement, then it's better to rely on the first solution using Character.isDigit().

Avoid solving this problem via parseInt() or parseLong(). First of all, it's bad practice to catch NumberFormatException and take business logic decisions in the catch block. Second, these methods verify whether the string is a valid number, not whether it contains only digits (for example, -4 is valid).  
For third-party library support, please consider the Apache Commons Lang, StringUtils.isNumeric().

5. Counting vowels and consonants

The following code is for English, but depending on how many languages you are covering, the number of vowels and consonants may differ and the code should be adjusted accordingly.

The first solution to this problem requires traversing the string characters and doing the following:

1. We need to check whether the current character is a vowel (this is convenient since we only have five pure vowels in English; other languages have more vowels, but the number is still small).
2. If the current character is not a vowel, then check whether it sits between 'a' and 'z' (this means that the current character is a consonant).

Notice that, initially, the given String object is transformed into lowercase. This is useful to avoid comparisons with uppercase characters. For example, the comparison is accomplished only against 'a' instead of 'A' and 'a'.

The code for this solution is as follows:

private static final Set<Character> allVowels  
 = new HashSet(Arrays.asList('a', 'e', 'i', 'o', 'u'));  
  
public static Pair<Integer, Integer>   
 countVowelsAndConsonants(String str) {  
  
 str = str.toLowerCase();  
 int vowels = 0;  
 int consonants = 0;  
  
 for (int i = 0; i < str.length(); i++) {  
 char ch = str.charAt(i);  
 if (allVowels.contains(ch)) {  
 vowels++;  
 } else if ((ch >= 'a' && ch <= 'z')) {  
 consonants++;  
 }  
 }  
  
 return Pair.of(vowels, consonants);  
}Copy

In Java 8 functional style, this code can be rewritten using chars() and filter():

private static final Set<Character> allVowels  
 = new HashSet(Arrays.asList('a', 'e', 'i', 'o', 'u'));  
  
public static Pair<Long, Long> countVowelsAndConsonants(String str) {  
  
 str = str.toLowerCase();  
  
 long vowels = str.chars()  
 .filter(c -> allVowels.contains((char) c))  
 .count();  
  
 long consonants = str.chars()  
 .filter(c -> !allVowels.contains((char) c))  
 .filter(ch -> (ch >= 'a' && ch<= 'z'))  
 .count();  
  
 return Pair.of(vowels, consonants);  
}Copy

The given string is filtered accordingly and the count() terminal operation returns the result. Relying on partitioningBy() will reduce the code, as follows:

Map<Boolean, Long> result = str.chars()  
 .mapToObj(c -> (char) c)  
 .filter(ch -> (ch >= 'a' && ch <= 'z'))  
 .collect(partitioningBy(c -> allVowels.contains(c), counting()));  
  
return Pair.of(result.get(true), result.get(false));Copy

Done! Now, let's see how we can count occurrences of a certain character in a string.

6. Counting the occurrences of a certain character

A simple solution to this problem consists of the following two steps:

1. Replace every occurrence of the character in the given string with "" (basically, this is like removing all of the occurrences of this character in the given string).
2. Subtract the length of the string that was obtained in the first step from the length of the initial string.

The code for this method is as follows:

public static int countOccurrencesOfACertainCharacter(  
 String str, char ch) {  
  
 return str.length() - str.replace(String.valueOf(ch), "").length();  
}Copy

The following solution covers Unicode surrogate pairs as well:

public static int countOccurrencesOfACertainCharacter(  
 String str, String ch) {   
  
 if (ch.codePointCount(0, ch.length()) > 1) {  
 // there is more than 1 Unicode character in the given String  
 return -1;   
 }  
  
 int result = str.length() - str.replace(ch, "").length();  
  
 // if ch.length() return 2 then this is a Unicode surrogate pair  
 return ch.length() == 2 ? result / 2 : result;  
}Copy

Another easy to implement and fast solution consists of looping the string characters (a single traversal) and comparing each character with the given character. Increase the counter by one for every match:

public static int countOccurrencesOfACertainCharacter(  
 String str, char ch) {  
  
 int count = 0;  
  
 for (int i = 0; i < str.length(); i++) {  
 if (str.charAt(i) == ch) {  
 count++;  
 }  
 }  
  
 return count;  
}Copy

The solution that covers the Unicode surrogate pairs is in the code that's bundled with this book. In Java 8 functional style, one solution consists of using filter() or reduce(). For example, using filter() will result in the following code:

public static long countOccurrencesOfACertainCharacter(  
 String str, char ch) {  
  
 return str.chars()  
 .filter(c -> c == ch)  
 .count();  
}Copy

The solution that covers the Unicode surrogate pairs is in the code that's bundled with this book.

For third-party library support, please consider Apache Commons Lang, StringUtils.countMatches(), Spring Framework, StringUtils.countOccurrencesOf(), and Guava, CharMatcher.is().countIn().

7. Converting a string into an int, long, float, or double

Let's consider the following strings (negatives can be used as well):

private static final String TO\_INT = "453";   
private static final String TO\_LONG = "45234223233";   
private static final String TO\_FLOAT = "45.823F";  
private static final String TO\_DOUBLE = "13.83423D";Copy

A proper solution for converting String into int, long, float, or double consists of using the following Java methods of the Integer, Long, Float, and Double classes—parseInt(), parseLong(), parseFloat(), and parseDouble():

int toInt = Integer.parseInt(TO\_INT);  
long toLong = Long.parseLong(TO\_LONG);  
float toFloat = Float.parseFloat(TO\_FLOAT);  
double toDouble = Double.parseDouble(TO\_DOUBLE);Copy

Converting String into an Integer, Long, Float, or Double object can be accomplished via the following Java methods—Integer.valueOf(), Long.valueOf(), Float.valueOf(), and Double.valueOf():

Integer toInt = Integer.valueOf(TO\_INT);  
Long toLong = Long.valueOf(TO\_LONG);  
Float toFloat = Float.valueOf(TO\_FLOAT);  
Double toDouble = Double.valueOf(TO\_DOUBLE);Copy

When a String cannot be converted successfully, Java throws a NumberFormatException exception. The following code speaks for itself:

private static final String WRONG\_NUMBER = "452w";  
  
try {  
 Integer toIntWrong1 = Integer.valueOf(WRONG\_NUMBER);  
} catch (NumberFormatException e) {  
 System.err.println(e);  
 // handle exception  
}  
  
try {  
 int toIntWrong2 = Integer.parseInt(WRONG\_NUMBER);  
} catch (NumberFormatException e) {  
 System.err.println(e);  
 // handle exception  
}Copy

For third-party library support, please consider Apache Commons BeanUtils: IntegerConverter, LongConverter, FloatConverter, and DoubleConverter.

8. Removing white spaces from a string

The solution to this problem consists of using the String.replaceAll() method with the \s regular expression. Mainly, \s removes all white spaces, including the non-visible ones, such as \t, \n, and \r:

public static String removeWhitespaces(String str) {  
 return str.replaceAll("\\s", "");  
}Copy

Starting with JDK 11, String.isBlank() checks whether the string is empty or contains only white space code points. For third-party library support, please consider Apache Commons Lang, StringUtils.deleteWhitespace(), and the Spring Framework, StringUtils.trimAllWhitespace().

9. Joining multiple strings with a delimiter

There are several solutions that fit well and solve this problem. Before Java 8, a convenient approach relied on StringBuilder, as follows:

public static String joinByDelimiter(char delimiter, String...args) {  
  
 StringBuilder result = new StringBuilder();  
  
 int i = 0;  
 for (i = 0; i < args.length - 1; i++) {  
 result.append(args[i]).append(delimiter);  
 }  
 result.append(args[i]);  
  
 return result.toString();  
}Copy

Starting with Java 8, there are at least three more solutions to this problem. One of these solutions relies on the StringJoiner utility class. This class can be used to construct a sequence of characters separated by a delimiter (for example, a comma).

It supports an optional prefix and suffix as well (ignored here):

public static String joinByDelimiter(char delimiter, String...args) {  
 StringJoiner joiner = new StringJoiner(String.valueOf(delimiter));  
  
 for (String arg: args) {  
 joiner.add(arg);  
 }  
  
 return joiner.toString();  
}Copy

Another solution relies on the String.join() method. This method was introduced in Java 8 and comes in two flavors:

String join​(CharSequence delimiter, CharSequence... elems)  
String join​(CharSequence delimiter,  
 Iterable<? extends CharSequence> elems)Copy

An example of joining several strings delimited by a space is as follows:

String result = String.join(" ", "how", "are", "you"); // how are youCopy

Going further, Java 8 streams and Collectors.joining() can be useful as well:

public static String joinByDelimiter(char delimiter, String...args) {  
 return Arrays.stream(args, 0, args.length)  
 .collect(Collectors.joining(String.valueOf(delimiter)));  
}Copy

Pay attention to concatenating strings via the += operator, and the concat() and String.format() methods. These can be used to join several strings, but they are prone to performance penalties. For example, the following code relies on += and is much slower than relying on StringBuilder :  
  
String str = "";  
for(int i = 0; i < 1\_000\_000; i++) {  
str += "x";  
}  
  
+= is appended to a string and reconstructs a new string, and that costs time.  
  
For third-party library support, please consider Apache Commons Lang, StringUtils.join(), and Guava, Joiner.

10. Generating all permutations

Problems that involve permutations commonly involve *recursivity* as well. Basically, recursivity is defined as a process where some initial state is given and each *successive state* is defined in terms of the *preceding state*.

In our case, the state can be materialized by the letters of the given string. The initial state contains the initial string and each successive state can be computed by the following formula—each letter of the string will become the first letter of the string (swap positions) and then permute all of the remaining letters using a recursive call. While non-recursive or other recursive solutions exist, this is a classical solution to this problem.

Representing this solution for a string, ABC, can be done like so (notice how permutations are done):

Diagrama

Descripción generada automáticamente

Coding this algorithm will result in something like the following:

public static void permuteAndPrint(String str) {  
  
 permuteAndPrint("", str);  
}  
  
private static void permuteAndPrint(String prefix, String str) {  
  
 int n = str.length();  
  
 if (n == 0) {  
 System.out.print(prefix + " ");  
 } else {  
 for (int i = 0; i < n; i++) {  
 permuteAndPrint(prefix + str.charAt(i),  
 str.substring(i + 1, n) + str.substring(0, i));  
 }  
 }  
}Copy

Initially, the prefix should be an empty string, "". At each iteration, the prefix will concatenate (fix) the next letter from the string. The remaining letters are passed through the method again.

Let's suppose that this method lives in a utility class named Strings. You can call it like so:

Strings.permuteAndStore("ABC");Copy

This will produce the following output:

ABC ACB BCA BAC CAB CBACopy

Notice that this solution prints the result on the screen. Storing the result implies adding Set to the implementation. It is preferable to use Set since it eliminates duplicates:

public static Set<String> permuteAndStore(String str) {  
  
 return permuteAndStore("", str);  
}  
  
private static Set<String>   
 permuteAndStore(String prefix, String str) {  
  
 Set<String> permutations = new HashSet<>();  
 int n = str.length();  
  
 if (n == 0) {  
 permutations.add(prefix);  
 } else {  
 for (int i = 0; i < n; i++) {  
 permutations.addAll(permuteAndStore(prefix + str.charAt(i),  
 str.substring(i + 1, n) + str.substring(0, i)));  
 }  
 }  
  
 return permutations;  
}Copy

For example, if the passed string is TEST, then Set will cause the following output (these are all unique permutations):

ETST SETT TEST TTSE STTE STET TETS TSTE TSET TTES ESTT ETTSCopy

Using List instead of Set will result in the following output (notice the duplicates):

TEST TETS TSTE TSET TTES TTSE ESTT ESTT ETTS ETST ETST ETTS STTE STET STET STTE SETT SETT TTES TTSE TEST TETS TSTE TSETCopy

There are 24 permutations. It is easy to determine the number of resulted permutations by computing the *n* factorial (*n!*). For *n=4* (length of the string), *4! = 1 x 2 x 3 x 4 = 24*. When expressed in recursive style, this is *n! = n x (n-1)!*.

Since *n!* results in high numbers extremely fast (example, *10! = 3628800*), it is advisable to avoid storing the results. For a 10-character string such as HELICOPTER, there are 3,628,800 permutations!

Trying to implement this solution in Java 8 functional style will result in something like the following:

private static void permuteAndPrintStream(String prefix, String str) {  
  
 int n = str.length();  
  
 if (n == 0) {  
 System.out.print(prefix + " ");  
 } else {  
 IntStream.range(0, n)  
 .parallel()  
 .forEach(i -> permuteAndPrintStream(prefix + str.charAt(i),  
 str.substring(i + 1, n) + str.substring(0, i)));  
 }  
}Copy

As a bonus, a solution that returns Stream<String> is available in the code bundled with this book.

11. Checking whether a string is a palindrome

Just as a quick reminder, a *palindrome* (whether a string or a number) looks unchanged when it's reversed. This means that processing (reading) a palindrome can be done from both directions and the same result will be obtained (for example, the word *madam* is a palindrome, while the word *madame* is not).

An easy to implement solution consists of comparing the letters of the given string in a *meet-in-the-middle* approach. Basically, this solution compares the first character with the last one, the second character with the last by one, and so on until the middle of the string is reached. The implementation relies on the while statement:

public static boolean isPalindrome(String str) {  
  
 int left = 0;  
 int right = str.length() - 1;  
  
 while (right > left) {  
 if (str.charAt(left) != str.charAt(right)) {  
 return false;  
 }  
  
 left++;  
 right--;  
 }  
 return true;  
}Copy

Rewriting the preceding solution in a more concise approach will consist of relying on a for statement instead of a while statement, as follows:

public static boolean isPalindrome(String str) {  
  
 int n = str.length();  
  
 for (int i = 0; i < n / 2; i++) {  
 if (str.charAt(i) != str.charAt(n - i - 1)) {  
 return false;  
 }  
 }  
 return true;  
}Copy

But can this solution be reduced to a single line of code? The answer is yes.

The Java API provides the StringBuilder class, which uses the reverse() method. As its name suggests, the reverse() method returns the reverse given string. In the case of a palindrome, the given string should be equal to the reverse version of it:

public static boolean isPalindrome(String str) {  
  
 return str.equals(new StringBuilder(str).reverse().toString());  
}Copy

In Java 8 functional style, there is a single line of code for this as well. Simply define IntStream ranging from 0 to half of the given string and use the noneMatch() *short-circuiting* terminal operation with a predicate that compares the letters by following the *meet-in-the-middle* approach:

public static boolean isPalindrome(String str) {  
  
 return IntStream.range(0, str.length() / 2)  
 .noneMatch(p -> str.charAt(p) !=   
 str.charAt(str.length() - p - 1));  
}Copy

Now, let's talk about removing duplicate characters from the given string.

12. Removing duplicate characters

Let's start with a solution to this problem that relies on StringBuilder. Mainly, the solution should loop the characters of the given string and construct a new string containing unique characters (it is not possible to simply remove characters from the given string since, in Java, a string is immutable).

The StringBuilder class exposes a method named indexOf(), which returns the index within the given string of the first occurrence of the specified substring (in our case, the specified character). So, a potential solution to this problem would be to loop the characters of the given string and add them one by one in StringBuilder every time the indexOf() method that's applied to the current character returns -1 (this negative means that StringBuilder doesn't contain the current character):

public static String removeDuplicates(String str) {  
  
 char[] chArray = str.toCharArray(); // or, use charAt(i)  
 StringBuilder sb = new StringBuilder();  
  
 for (char ch : chArray) {  
 if (sb.indexOf(String.valueOf(ch)) == -1) {  
 sb.append(ch);  
 }  
 }  
 return sb.toString();  
}Copy

The next solution relies on a collaboration between HashSet and StringBuilder. Mainly, HashSet ensures that duplicates are eliminated, while StringBuilder stores the resulting string. If HashSet.add() returns true, then we add the character in StringBuilder as well:

public static String removeDuplicates(String str) {  
  
 char[] chArray = str.toCharArray();  
 StringBuilder sb = new StringBuilder();  
 Set<Character> chHashSet = new HashSet<>();  
  
 for (char c: chArray) {  
 if (chHashSet.add(c)) {  
 sb.append(c);  
 }  
 }  
 return sb.toString();  
}Copy

The solutions we've presented so far use the toCharArray() method to convert the given string into char[]. Alternatively, both solutions can use str.charAt(position) as well.

The third solution relies on Java 8 functional style:

public static String removeDuplicates(String str) {  
  
 return Arrays.asList(str.split("")).stream()  
 .distinct()  
 .collect(Collectors.joining());  
}Copy

First, the solution converts the given string into Stream<String>, where each entry is actually a single character. Furthermore, the solution applies the stateful intermediate operation, distinct(). This operation will eliminate duplicates from the stream, so it returns a stream without duplicates. Finally, the solution calls the collect() terminal operation and relies on Collectors.joining(), which simply concatenates the characters into a string in the encounter order.

13. Removing a given character

A solution that relies on JDK support can exploit the String.replaceAll() method. This method replaces each substring (in our case, each character) of the given string that matches the given regular expression (in our case, the regular expression is the character itself) with the given replacement (in our case, the replacement is an empty string, ""):

public static String removeCharacter(String str, char ch) {  
  
 return str.replaceAll(Pattern.quote(String.valueOf(ch)), "");  
}Copy

Notice that the regular expression is wrapped in the Pattern.quote() method. This is needed to escape special characters such as <, (, [, {, \, ^, -, =, $, !, |, ], }, ), ?, \*, +, ., and >. Mainly, this method returns a literal pattern string for the specified string.

Now, let's take a look at a solution that avoids regular expressions. This time, the solution relies on StringBuilder. Basically, the solution loops the characters of the given string and compares each character with the character to remove. Each time the current character is different from the character to remove, the current character is appended in StringBuilder:

public static String removeCharacter(String str, char ch) {  
  
 StringBuilder sb = new StringBuilder();  
 char[] chArray = str.toCharArray();  
  
 for (char c : chArray) {  
 if (c != ch) {  
 sb.append(c);  
 }  
 }  
  
 return sb.toString();  
}Copy

Finally, let's focus on a Java 8 functional style approach. This is a four-step approach:

1. Convert the string into IntStream via the String.chars() method
2. Filter IntStream to eliminate duplicates
3. Map the resulted IntStream to Stream<String>
4. Join the strings from this stream and collect them as a single string

The code for this solution can be written as follows:

public static String removeCharacter(String str, char ch) {  
  
 return str.chars()  
 .filter(c -> c != ch)  
 .mapToObj(c -> String.valueOf((char) c))  
 .collect(Collectors.joining());  
}Copy

Alternatively, if we want to remove a Unicode surrogate pair, then we can rely on codePointAt() and codePoints(), as shown in the following implementation:

public static String removeCharacter(String str, String ch) {  
   
 int codePoint = ch.codePointAt(0);  
  
 return str.codePoints()  
 .filter(c -> c != codePoint)  
 .mapToObj(c -> String.valueOf(Character.toChars(c)))  
 .collect(Collectors.joining());  
 }Copy

For third-party library support, please consider Apache Commons Lang, StringUtils.remove().

Now, let's talk about how to find the character with the most appearances.

14. Finding the character with the most appearances

A pretty straightforward solution relies on HashMap. This solution consists of three steps:

1. First, loop the characters of the given string and put the pairs of the key-value in HashMap where the key is the current character and the value is the current number of occurrences
2. Second, compute the maximum value in HashMap (for example, using Collections.max()) representing the maximum number of occurrences
3. Finally, get the character that has the maximum number of occurrences by looping the HashMap entry set

The utility method returns Pair<Character, Integer> containing the character with the most appearances and the number of appearances (notice that the white spaces are ignored). If you don't prefer to have this extra class, that is, Pair, then just rely on Map.Entry<K, V>:

public static Pair<Character, Integer> maxOccurenceCharacter(  
 String str) {  
  
 Map<Character, Integer> counter = new HashMap<>();  
 char[] chStr = str.toCharArray();  
  
 for (int i = 0; i < chStr.length; i++) {  
 char currentCh = chStr[i];  
 if (!Character.isWhitespace(currentCh)) { // ignore spaces  
 Integer noCh = counter.get(currentCh);  
 if (noCh == null) {  
 counter.put(currentCh, 1);  
 } else {  
 counter.put(currentCh, ++noCh);  
 }  
 }  
 }  
  
 int maxOccurrences = Collections.max(counter.values());  
 char maxCharacter = Character.MIN\_VALUE;  
  
 for (Entry<Character, Integer> entry: counter.entrySet()) {  
 if (entry.getValue() == maxOccurrences) {  
 maxCharacter = entry.getKey();  
 }  
 }  
  
 return Pair.of(maxCharacter, maxOccurrences);  
}Copy

If using HashMap looks cumbersome, then another solution (that's a little faster) consists of relying on the ASCII codes. This solution starts with an empty array of 256 indexes (256 is the maximum number of extended ASCII table codes; more information can be found in the *Finding the first non-repeated character* section). Furthermore, this solution loops the characters of the given string and keeps track of the number of appearances for each character by increasing the corresponding index in this array:

private static final int EXTENDED\_ASCII\_CODES = 256;  
...  
public static Pair<Character, Integer> maxOccurenceCharacter(  
 String str) {  
  
 int maxOccurrences = -1;  
 char maxCharacter = Character.MIN\_VALUE;  
 char[] chStr = str.toCharArray();  
 int[] asciiCodes = new int[EXTENDED\_ASCII\_CODES];  
  
 for (int i = 0; i < chStr.length; i++) {  
 char currentCh = chStr[i];  
 if (!Character.isWhitespace(currentCh)) { // ignoring space  
 int code = (int) currentCh;  
 asciiCodes[code]++;  
 if (asciiCodes[code] > maxOccurrences) {  
 maxOccurrences = asciiCodes[code];  
 maxCharacter = currentCh;  
 }  
 }  
 }  
  
 return Pair.of(maxCharacter, maxOccurrences);  
}Copy

The last solution we will discuss here relies on Java 8 functional style:

public static Pair<Character, Long>   
 maxOccurenceCharacter(String str) {  
  
 return str.chars()  
 .filter(c -> Character.isWhitespace(c) == false) // ignoring space  
 .mapToObj(c -> (char) c)  
 .collect(groupingBy(c -> c, counting()))  
 .entrySet()  
 .stream()  
 .max(comparingByValue())  
 .map(p -> Pair.of(p.getKey(), p.getValue()))  
 .orElse(Pair.of(Character.MIN\_VALUE, -1L));  
}Copy

To start, this solution collects distinct characters as keys in Map, along with their number of occurrences as values. Furthermore, it uses the Java 8 Map.Entry.comparingByValue() and max() terminal operations to determine the entry in the map with the highest value (highest number of occurrences). Since max() is a terminal operation, the solution may return Optional<Entry<Character, Long>>, but this solution adds an extra step and maps this entry to Pair<Character, Long>.

15. Sorting an array of strings by length

The first thing that comes to mind when sorting is the use of a comparator.

In this case, the solution should compare lengths of strings, and so the integers are returned by calling String.length() for each string in the given array. So, if the integers are sorted (ascending or descending), then the strings will be sorted.

The Java Arrays class already provides a sort() method that takes the array to sort and a comparator. In this case, Comparator<String> should do the job.

Before Java 7, code that implemented a comparator relied on the compareTo() method. Common usage of this method was to compute a difference of the *x1*-*x2* type, but this computation may lead to overflows. This makes compareTo() rather tedious. Starting with Java 7, Integer.compare() is the way to go (no overflow risks).

The following is a method that sorts the given array by relying on the Arrays.sort() method:

public static void sortArrayByLength(String[] strs, Sort direction) {  
 if (direction.equals(Sort.ASC)) {  
 Arrays.sort(strs, (String s1, String s2)   
 -> Integer.compare(s1.length(), s2.length()));  
 } else {  
 Arrays.sort(strs, (String s1, String s2)   
 -> (-1) \* Integer.compare(s1.length(), s2.length()));  
 }  
}Copy

Each wrapper of a primitive numeric type has a compare() method.

Starting with Java 8, the Comparator interface was enriched with a significant number of useful methods. One of these methods is comparingInt(), which takes a function that extracts an int sort key from the generic type and returns a Comparator<T> value that compares it with that sort key. Another useful method is reversed(), which reverses the current Comparator value.

Based on these two methods, we can empower Arrays.sort() as follows:

public static void sortArrayByLength(String[] strs, Sort direction) {  
 if (direction.equals(Sort.ASC)) {  
 Arrays.sort(strs, Comparator.comparingInt(String::length));  
 } else {  
 Arrays.sort(strs,   
 Comparator.comparingInt(String::length).reversed());  
 }  
}Copy

Comparators can be chained with the thenComparing() method.

The solutions we've presented here return void, which means that they sort the given array. To return a new sorted array and not alter the given array, we can use Java 8 functional style, as shown in the following snippet of code:

public static String[] sortArrayByLength(String[] strs,   
 Sort direction) {  
  
 if (direction.equals(Sort.ASC)) {  
 return Arrays.stream(strs)  
 .sorted(Comparator.comparingInt(String::length))  
 .toArray(String[]::new);  
 } else {  
 return Arrays.stream(strs)  
 .sorted(Comparator.comparingInt(String::length).reversed())  
 .toArray(String[]::new);  
 }  
}Copy

So, the code creates a stream from the given array, sorts it via the sorted() stateful intermediate operation, and collects the result in another array.

16. Checking that a string contains a substring

A very simple, one line of code solution relies on the String.contains() method.

This method returns a boolean value indicating whether the given substring is present in the string or not:

String text = "hello world!";  
String subtext = "orl";  
  
// pay attention that this will return true for subtext=""  
boolean contains = text.contains(subtext);Copy

Alternatively, a solution can be implemented by relying on String.indexOf() (or String.lastIndexOf()), as follows:

public static boolean contains(String text, String subtext) {  
  
 return text.indexOf(subtext) != -1; // or lastIndexOf()  
}Copy

Another solution can be implemented based on a regular expression, as follows:

public static boolean contains(String text, String subtext) {  
  
 return text.matches("(?i).\*" + Pattern.quote(subtext) + ".\*");  
}Copy

Notice that the regular expression is wrapped in the Pattern.quote() method. This is needed to escape special characters such as <([{\^-=$!|]})?\*+.> in the given substring.

For third-party library support, please consider Apache Commons Lang, StringUtils.containsIgnoreCase().

17. Counting substring occurrences in a string

Counting the number of occurrences of a string in another string is a problem that can have at least two interpretations:

* 11 in 111 occurs 1 time
* 11 in 111 occurs 2 times

In the first case (11 in 111 occurs 1 time), the solution can rely on the String.indexOf() method. One of the flavors of this method allows us to obtain the index within this string of the first occurrence of the specified substring, starting at the specified index (or -1, if there is no such occurrence). Based on this method, the solution can simply traverse the given string and count the given substring occurrences. The traversal starts from position 0 and continues until the substring is not found:

public static int countStringInString(String string, String toFind) {  
  
 int position = 0;  
 int count = 0;  
 int n = toFind.length();  
  
 while ((position = string.indexOf(toFind, position)) != -1) {  
 position = position + n;  
 count++;  
 }  
  
 return count;  
}Copy

Alternatively, the solution can use the String.split() method. Basically, the solution can split the given string using the given substring as a delimiter. The length of the resulting String[] array should be equal to the number of expected occurrences:

public static int countStringInString(String string, String toFind) {  
  
 int result = string.split(Pattern.quote(toFind), -1).length - 1;  
  
 return result < 0 ? 0 : result;  
}Copy

In the second case (11 in 111 occurs 2 times), the solution can rely on the Pattern and Matcher classes in a simple implementation, as follows:

public static int countStringInString(String string, String toFind) {  
  
 Pattern pattern = Pattern.compile(Pattern.quote(toFind));  
 Matcher matcher = pattern.matcher(string);  
  
 int position = 0;  
 int count = 0;  
  
 while (matcher.find(position)) {  
  
 position = matcher.start() + 1;  
 count++;  
 }  
  
 return count;  
}Copy

Nice! Let's continue with another problem with strings.

18. Checking whether two strings are anagrams

Two strings that have the same characters, but that are in a different order, are anagrams. Some definitions impose that anagrams are case-insensitive and/or that white spaces (blanks) should be ignored.

So, independent of the applied algorithm, the solution must convert the given string into lowercase and remove white spaces (blanks). Besides that, the first solution we mentioned sorts the arrays via Arrays.sort() and will check their equality via Arrays.equals().

Once they are sorted, if they are anagrams, they will be equal (the following diagram shows two words that are anagrams):

Forma

Descripción generada automáticamente con confianza media

This solution (including its Java 8 functional style version) is available in the code bundled with this book. The main drawback of these two solutions is represented by the sorting part. The following solution eliminates this step and relies on an empty array (initially containing only 0) of 256 indexes (extended ASCII table codes of characters—more information can be found in the *Finding the first non-repeated character* section).

The algorithm is pretty simple:

* For each character from the first string, this solution increases the value in this array corresponding to the ASCII code by 1
* For each character from the second string, this solution decreases the value in this array corresponding to the ASCII code by 1

The code is as follows:

private static final int EXTENDED\_ASCII\_CODES = 256;  
...  
public static boolean isAnagram(String str1, String str2) {  
  
 int[] chCounts = new int[EXTENDED\_ASCII\_CODES];  
 char[] chStr1 = str1.replaceAll("\\s",   
 "").toLowerCase().toCharArray();  
 char[] chStr2 = str2.replaceAll("\\s",   
 "").toLowerCase().toCharArray();  
  
 if (chStr1.length != chStr2.length) {  
 return false;  
 }  
  
 for (int i = 0; i < chStr1.length; i++) {  
 chCounts[chStr1[i]]++;  
 chCounts[chStr2[i]]--;  
 }  
  
 for (int i = 0; i < chCounts.length; i++) {  
 if (chCounts[i] != 0) {  
 return false;  
 }  
 }  
  
 return true;  
}Copy

At the end of this traversal, if the given strings are anagrams, then this array contains only 0.

19. Declaring multiline strings (text blocks)

At the time of writing this book, JDK 12 had a proposal for adding multiline strings known as *JEP 326: Raw String Literals*. But this was dropped at the last minute.

Starting with JDK 13, the idea was reconsidered and, unlike the declined raw string literals, text blocks are surrounded by three double quotes, """, as follows:

String text = """My high school,  
the Illinois Mathematics and Science Academy,  
showed me that anything is possible  
and that you're never too young to think big.""";Copy

Text blocks can be very useful for writing multiline SQL statements, using polyglot languages, and so on. More details can be found at <https://openjdk.java.net/jeps/355>.

Nevertheless, there are several surrogate solutions that can be used before JDK 13. These solutions have a common point—the use of the line separator:

private static final String LS = System.lineSeparator();Copy

Starting with JDK 8, a solution may rely on String.join(), as follows:

String text = String.join(LS,  
 "My high school, ",  
 "the Illinois Mathematics and Science Academy,",  
 "showed me that anything is possible ",  
 "and that you're never too young to think big.");Copy

Before JDK 8, an elegant solution may have relied on StringBuilder. This solution is available in the code bundled with this book.

While the preceding solutions are good fits for a relatively large number of strings, the following two are okay if we just have a few strings. The first one uses the + operator:

String text = "My high school, " + LS +  
 "the Illinois Mathematics and Science Academy," + LS +  
 "showed me that anything is possible " + LS +  
 "and that you're never too young to think big.";Copy

The second one uses String.format():

String text = String.format("%s" + LS + "%s" + LS + "%s" + LS + "%s",  
 "My high school, ",  
 "the Illinois Mathematics and Science Academy,",  
 "showed me that anything is possible ",  
 "and that you're never too young to think big.");Copy

How can we process each line of a multiline string? Well, a quick approach requires JDK 11, which comes with the String.lines() method. This method splits the given string via a line separator (which supports \n, \r, and \r\n) and transforms it into Stream<String>. Alternatively, the String.split() method can be used as well (this is available starting with JDK 1.4). If the number of strings becomes significant, it is advised to put them in a file and read/process them one by one (for example, via the getResourceAsStream() method). Other approaches rely on StringWriter or BufferedWriter.newLine().  
  
For third-party library support, please consider Apache Commons Lang, StringUtils.join(), Guava, Joiner, and the custom annotation, @Multiline.

20. Concatenating the same string n times

Before JDK 11, a solution could be quickly provided via StringBuilder, as follows:

public static String concatRepeat(String str, int n) {  
  
 StringBuilder sb = new StringBuilder(str.length() \* n);  
  
 for (int i = 1; i <= n; i++) {  
 sb.append(str);  
 }  
  
 return sb.toString();  
}Copy

Starting with JDK 11, the solution relies on the String.repeat(int count) method. This method returns a string resulting from concatenating this string count times. Behind the scenes, this method uses System.arraycopy(), which makes this very fast:

String result = "hello".repeat(5);Copy

Other solutions that can fit well in different scenarios are listed as follows:

* Following is a String.join()-based solution:

String result = String.join("", Collections.nCopies(5, TEXT));Copy

* Following is a Stream.generate()-based solution:

String result = Stream.generate(() -> TEXT)  
 .limit(5)  
 .collect(joining());Copy

* Following is a String.format()-based solution:

String result = String.format("%0" + 5 + "d", 0)  
 .replace("0", TEXT);Copy

* Following is a char[] based solution:

String result = new String(new char[5]).replace("\0", TEXT);Copy

For third-party library support, please consider Apache Commons Lang, StringUtils.repeat(), and Guava, Strings.repeat().

To check whether a string is a sequence of the same substring, rely on the following method:

public static boolean hasOnlySubstrings(String str) {  
  
 StringBuilder sb = new StringBuilder();  
  
 for (int i = 0; i < str.length() / 2; i++) {  
 sb.append(str.charAt(i));  
 String resultStr = str.replaceAll(sb.toString(), "");  
 if (resultStr.length() == 0) {  
 return true;  
 }  
 }  
  
 return false;  
}Copy

The solution loops half of the given string and progressively replaces it with "", a substring build, by appending the original string in StringBuilder, character by character. If these replacements result in an empty string, it means that the given string is a sequence of the same substring.

21. Removing leading and trailing spaces

The quickest solution to this problem probably relies on the String.trim() method. This method is capable of removing all leading and trailing spaces, that is, any character whose code point is less than or equal to U+0020 or 32 (the space character):

String text = "\n \n\n hello \t \n \r";  
String trimmed = text.trim();Copy

The preceding snippet of code will work as expected. The trimmed string will be hello. This only works because all of the white spaces that are being used are less than U+0020 or 32 (the space character). There are 25 characters (<https://en.wikipedia.org/wiki/Whitespace_character#Unicode>) defined as white spaces and trim() covers only a part of them (in short, trim() is not Unicode aware). Let's consider the following string:

char space = '\u2002';  
String text = space + "\n \n\n hello \t \n \r" + space;Copy

\u2002 is another type of white space that trim() doesn't recognize (\u2002 is above \u0020). This means that, in such cases, trim() will not work as expected. Starting with JDK 11, this problem has a solution named strip(). This method extends the power of trim() into the land of Unicode:

String stripped = text.strip();Copy

This time, all of the leading and trailing white spaces are removed.

Moreover, JDK 11 comes with two flavors of strip() for removing only the leading (stripLeading()) or only the trailing (stripTrailing()) white spaces. The trim() method doesn't have these flavors.

22. Finding the longest common prefix

Let's consider the following array of strings:

String[] texts = {"abc", "abcd", "abcde", "ab", "abcd", "abcdef"};Copy

Now, let's put these strings one below the other, as follows:

**ab**c  
**ab**cd  
**ab**cde  
**ab  
ab**cd  
**ab**cdefCopy

A simple comparison of these strings reveals that ab is the longest common prefix. Now, let's dive into a solution for solving this problem. The solution that we've presented here relies on a straightforward comparison. This solution takes the first string from the array and compares each of its characters in the rest of the strings. The algorithm stops if either of the following happens:

* The length of the first string is greater than the length of any of the other strings
* The current character of the first string is not the same as the current character of any of the other strings

If the algorithm forcibly stops because of one of the preceding scenarios, then the longest common prefix is the substring from 0 to the index of the current character from the first string. Otherwise, the longest common prefix is the first string from the array. The code for this solution is as follows:

public static String longestCommonPrefix(String[] strs) {  
  
 if (strs.length == 1) {  
 return strs[0];  
 }  
  
 int firstLen = strs[0].length();  
  
 for (int prefixLen = 0; prefixLen < firstLen; prefixLen++) {  
 char ch = strs[0].charAt(prefixLen);  
 for (int i = 1; i < strs.length; i++) {  
 if (prefixLen >= strs[i].length()   
 || strs[i].charAt(prefixLen) != ch) {  
 return strs[i].substring(0, prefixLen);  
 }  
 }  
 }  
  
 return strs[0];  
}Copy

Other solutions to this problem use well-known algorithms such as **Binary Search** or **Trie**. In the source code that accompanies this book, there is a solution based on Binary Search as well.

23. Applying indentation

Starting with JDK 12, we can indent text via the String.indent(int n) method.

Let's assume that we have the following String values:

String days = "Sunday\n"   
 + "Monday\n"   
 + "Tuesday\n"   
 + "Wednesday\n"   
 + "Thursday\n"   
 + "Friday\n"   
 + "Saturday";Copy

Printing this String values with an indentation of 10 spaces can be done as follows:

System.out.print(days.indent(10));Copy

The output will be as follows:

Texto

Descripción generada automáticamente

Now, let's try a cascade indentation:

List<String> days = Arrays.asList("Sunday", "Monday", "Tuesday",  
 "Wednesday", "Thursday", "Friday", "Saturday");  
  
for (int i = 0; i < days.size(); i++) {  
 System.out.print(days.get(i).indent(i));  
}Copy

The output will be as follows:

Texto, Carta

Descripción generada automáticamente

Now, let's indent depending on the length of the String value:

days.stream()  
 .forEachOrdered(d -> System.out.print(d.indent(d.length())));Copy

The output will be as follows:

Texto, Carta

Descripción generada automáticamente

How about indenting a piece of HTML code? Let's see:

String html = "<html>";  
String body = "<body>";  
String h2 = "<h2>";  
String text = "Hello world!";  
String closeH2 = "</h2>";  
String closeBody = "</body>";  
String closeHtml = "</html>";  
  
System.out.println(html.indent(0) + body.indent(4) + h2.indent(8)   
 + text.indent(12) + closeH2.indent(8) + closeBody.indent(4)  
 + closeHtml.indent(0));Copy

The output will be as follows:

Texto, Carta

Descripción generada automáticamente

24. Transforming strings

Let's assume that we have a string and we want to transform it into another string (for example, transform it into upper case). We can do this by applying a function such as Function<? super String,​ ? extends R>.

In JDK 8, we can accomplish this via map(), as shown in the following two simple examples:

// hello world  
String resultMap = Stream.of("hello")  
 .map(s -> s + " world")  
 .findFirst()  
 .get();  
  
// GOOOOOOOOOOOOOOOOL! GOOOOOOOOOOOOOOOOL!  
String resultMap = Stream.of("gooool! ")  
 .map(String::toUpperCase)  
 .map(s -> s.repeat(2))  
 .map(s -> s.replaceAll("O", "OOOO"))  
 .findFirst()  
 .get();Copy

Starting with JDK 12, we can rely on a new method named transform​(Function<? super String, ​? extends R> f). Let's rewrite the preceding snippets of code via transform():

// hello world  
String result = "hello".transform(s -> s + " world");  
  
// GOOOOOOOOOOOOOOOOL! GOOOOOOOOOOOOOOOOL!  
String result = "gooool! ".transform(String::toUpperCase)  
 .transform(s -> s.repeat(2))  
 .transform(s -> s.replaceAll("O", "OOOO"));Copy

While map() is more general, transform() is dedicated to applying a function to a string and returns the resulting string.

25. Computing the minimum and maximum of two numbers

Before JDK 8, a possible solution would be to rely on the Math.min() and Math.max() methods, as follows:

int i1 = -45;  
int i2 = -15;  
int min = Math.min(i1, i2);  
int max = Math.max(i1, i2);Copy

The Math class provides a min() and a max() method for each primitive numeric type (int, long, float, and double).

Starting with JDK 8, each wrapper class of primitive numeric types (Integer, Long, Float, and Double) comes with dedicated min() and max() methods, and, behind these methods, there are invocations of their correspondents from the Math class. See the following example (this is a little bit more expressive):

double d1 = 0.023844D;  
double d2 = 0.35468856D;  
double min = Double.min(d1, d2);  
double max = Double.max(d1, d2);Copy

In a functional style context, a potential solution will rely on the BinaryOperator functional interface. This interface comes with two methods, minBy() and maxBy():

float f1 = 33.34F;  
final float f2 = 33.213F;  
float min = BinaryOperator.minBy(Float::compare).apply(f1, f2);  
float max = BinaryOperator.maxBy(Float::compare).apply(f1, f2);Copy

These two methods are capable of returning the minimum (respectively, the maximum) of two elements according to the specified comparator.

26. Summing two large int/long values and operation overflow

Let's dive into the solution by starting with the + operator, as in the following example:

int x = 2;  
int y = 7;  
int z = x + y; // 9Copy

This is a very simple approach and works fine for most of the computations that involve int, long, float, and double.

Now, let's apply this operator on the following two large numbers (sum 2,147,483,647 with itself):

int x = Integer.MAX\_VALUE;  
int y = Integer.MAX\_VALUE;  
int z = x + y; // -2Copy

This time, z will be equal to -2, which is not the expected result, that is, 4,294,967,294. Changing only the z type from int to long will not help. However, changing the types of x and y from int to long as well *will* help:

long x = Integer.MAX\_VALUE;  
long y = Integer.MAX\_VALUE;  
long z = x + y; // 4294967294Copy

But the problem will reappear if, instead of Integer.MAX\_VALUE, there is Long.MAX\_VALUE:

long x = Long.MAX\_VALUE;  
long y = Long.MAX\_VALUE;  
long z = x + y; // -2Copy

Starting with JDK 8, the + operator has been wrapped in a more expressive way by each wrapper of a primitive numeric type. Therefore, the Integer, Long, Float, and Double classes have a sum() method:

long z = Long.sum(); // -2Copy

Behind the scenes, the sum() methods uses the + operator as well, so they simply produce the same result.

But also starting with JDK 8, the Math class was enriched with two addExact() methods. There is one addExact() for summing two int variables and one for summing two long variables. These methods are very useful if the result is prone to overflowing int or long, as shown in the preceding case. In such cases, these methods throw ArithmeticException instead of returning a misleading result, as in the following example:

int z = Math.addExact(x, y); // throw ArithmeticExceptionCopy

The code will throw an exception such as java.lang.ArithmeticException: integer overflow. This is useful since it allows us to avoid introducing misleading results in further computations (for example, earlier, -2 could silently enter further computations).

In a functional style context, a potential solution will rely on the BinaryOperator functional interface, as follows (simply define the operation of the two operands of the same type):

BinaryOperator<Integer> operator = Math::addExact;  
int z = operator.apply(x, y);Copy

Besides addExact(), Math has multiplyExact(), substractExact(), and negateExact(). Moreover, the well-known increment and decrement expressions, i++ and i--, can be controlled for overflowing their domains via the incrementExact() and decrementExact() methods (for example, Math.incrementExact(i)). Notice that these methods are only available for int and long.

When working with a large number, also focus on the BigInteger (immutable arbitrary-precision integers) and BigDecimal (immutable, arbitrary-precision signed decimal numbers) classes.

27. String as an unsigned number in the radix

The support for unsigned arithmetic was added to Java starting with version 8. The Byte, Short, Integer, and Long classes were affected the most by this addition.

In Java, strings representing positive numbers can be parsed as unsigned int and long types via the parseUnsignedInt() and parseUnsignedLong() JDK 8 methods. For example, let's consider the following integer as a string:

String nri = "255500";Copy

The solution to parsing it into an unsigned int value in the radix of 36 (the maximum accepted radix) looks as follows:

int result = Integer.parseUnsignedInt(nri, Character.MAX\_RADIX);Copy

The first argument is the number, while the second is the radix. The radix should be in the range [2, 36] or [Character.MIN\_RADIX, Character.MAX\_RADIX].

Using a radix of 10 can be easily accomplished as follows (this method applies a radix of 10 by default):

int result = Integer.parseUnsignedInt(nri);Copy

Starting with JDK 9, parseUnsignedInt() has a new flavor. Besides the string and the radix, this method accepts a range of the [beginIndex, endIndex] type. This time, the parsing is accomplished in this range. For example, specifying the range [1, 3] can be done as follows:

int result = Integer.parseUnsignedInt(nri, 1, 4, Character.MAX\_RADIX);Copy

The parseUnsignedInt() method can parse strings that represent numbers greater than Integer.MAX\_VALUE (trying to accomplish this via Integer.parseInt() will throw a java.lang.NumberFormatException exception):

// Integer.MAX\_VALUE + 1 = 2147483647 + 1 = 2147483648  
int maxValuePlus1 = Integer.parseUnsignedInt("2147483648");Copy

The same set of methods exist for long numbers in the Long class (for example, parseUnsignedLong()).

28. Converting into a number by an unsigned conversion

The problem requires that we convert the given signed int into long via an unsigned conversion. So, let's consider signed Integer.MIN\_VALUE, which is -2,147,483,648.

In JDK 8, by using the Integer.toUnsignedLong() method, the conversion will be as follows (the result will be 2,147,483,648):

long result = Integer.toUnsignedLong(Integer.MIN\_VALUE);Copy

Here is another example that converts the signed Short.MIN\_VALUE and Short.MAX\_VALUE into unsigned integers:

int result1 = Short.toUnsignedInt(Short.MIN\_VALUE);  
int result2 = Short.toUnsignedInt(Short.MAX\_VALUE);Copy

Other methods from the same category are Integer.toUnsignedString(), Long.toUnsignedString(), Byte.toUnsignedInt(), Byte.toUnsignedLong(), Short.toUnsignedInt(), and Short.toUnsignedLong().

29. Comparing two unsigned numbers

Let's consider two signed integers, Integer.MIN\_VALUE (-2,147,483,648) and Integer.MAX\_VALUE (2,147,483,647). Comparing these integers (signed values) will result in -2,147,483,648 being smaller than 2,147,483,647:

// resultSigned is equal to -1 indicating that  
// MIN\_VALUE is smaller than MAX\_VALUE  
int resultSigned = Integer.compare(Integer.MIN\_VALUE,   
 Integer.MAX\_VALUE);Copy

In JDK 8, these two integers can be compared as unsigned values via the Integer.compareUnsigned() method (this is the equivalent of Integer.compare() for unsigned values). Mainly, this method ignores the notion of *sign bit*, and the *left-most bit* is considered the most significant bit. Under the unsigned values umbrella, this method returns 0 if the compared numbers are equal, a value less than 0 if the first unsigned value is smaller than the second, and a value greater than 0 if the first unsigned value is greater than the second.

The following comparison returns 1, indicating that the unsigned value of Integer.MIN\_VALUE is greater than the unsigned value of Integer.MAX\_VALUE:

// resultSigned is equal to 1 indicating that  
// MIN\_VALUE is greater than MAX\_VALUE  
int resultUnsigned   
 = Integer.compareUnsigned(Integer.MIN\_VALUE, Integer.MAX\_VALUE);Copy

The compareUnsigned() method is available in the Integer and Long classes starting with JDK 8, and in the Byte and Short classes starting with JDK 9.

30. Division and modulo of unsigned values

Computing the unsigned quotient and remainder that resulted from the division of two unsigned values is supported by the JDK 8 unsigned arithmetic API via the divideUnsigned() and remainderUnsigned() methods.

Let's consider the Interger.MIN\_VALUE and Integer.MAX\_VALUE signed numbers and let's apply division and modulo. There's nothing new here:

// signed division  
// -1  
int divisionSignedMinMax = Integer.MIN\_VALUE / Integer.MAX\_VALUE;   
  
// 0  
int divisionSignedMaxMin = Integer.MAX\_VALUE / Integer.MIN\_VALUE;  
  
// signed modulo  
// -1  
int moduloSignedMinMax = Integer.MIN\_VALUE % Integer.MAX\_VALUE;   
  
// 2147483647  
int moduloSignedMaxMin = Integer.MAX\_VALUE % Integer.MIN\_VALUE; Copy

Now, let's treat Integer.MIN\_VALUE and Integer.MAX\_VALUE as unsigned values and let's apply divideUnsigned() and remainderUnsigned():

// division unsigned  
int divisionUnsignedMinMax = Integer.divideUnsigned(  
 Integer.MIN\_VALUE, Integer.MAX\_VALUE); // 1  
int divisionUnsignedMaxMin = Integer.divideUnsigned(  
 Integer.MAX\_VALUE, Integer.MIN\_VALUE); // 0  
  
// modulo unsigned  
int moduloUnsignedMinMax = Integer.remainderUnsigned(  
 Integer.MIN\_VALUE, Integer.MAX\_VALUE); // 1  
int moduloUnsignedMaxMin = Integer.remainderUnsigned(  
 Integer.MAX\_VALUE, Integer.MIN\_VALUE); // 2147483647Copy

Notice their similarity to the comparison operation. Both operations, that is, unsigned division and unsigned modulo, interpret all of the bits as *value bits* and ignore the *sign bit*.

divideUnsigned() and remainderUnsigned() are present in the Integer and Long classes, respectively.

31. double/float is a finite floating-point value

This problem arises from the fact that some floating-point methods and operations produce Infinity or NaN as results instead of throwing an exception.

The solution to checking whether the given float/double is a finite floating-point value relies on the following conditions—the absolute value of the given float/double value must not exceed the largest positive finite value of the float/double type:

// for float  
Math.abs(f) <= Float.MAX\_VALUE;  
  
// for double  
Math.abs(d) <= Double.MAX\_VALUECopy

Starting with Java 8, the preceding conditions were exposed via two dedicated flag-methods, Float.isFinite() and Double.isFinite(). Therefore, the following examples are valid test cases for finite floating-point values:

Float f1 = 4.5f;  
boolean f1f = Float.isFinite(f1); // f1 = 4.5, is finite  
  
Float f2 = f1 / 0;  
boolean f2f = Float.isFinite(f2); // f2 = Infinity, is not finite  
  
Float f3 = 0f / 0f;  
boolean f3f = Float.isFinite(f3); // f3 = NaN, is not finite  
  
Double d1 = 0.000333411333d;  
boolean d1f = Double.isFinite(d1); // d1 = 3.33411333E-4,is finite  
  
Double d2 = d1 / 0;  
boolean d2f = Double.isFinite(d2); // d2 = Infinity, is not finite  
  
Double d3 = Double.POSITIVE\_INFINITY \* 0;  
boolean d3f = Double.isFinite(d3); // d3 = NaN, is not finiteCopy

These methods are handy in conditions such as the following:

if (Float.isFinite(d1)) {  
 // do a computation with d1 finite floating-point value  
} else {  
 // d1 cannot enter in further computations  
}Copy

32. Applying logical AND/OR/XOR to two boolean expressions

The truth table of elementary logic operations (**AND**, **OR**, and **XOR**) looks as follows:

Tabla

Descripción generada automáticamente

In Java, the logical **AND** operator is represented as &&, the logical **OR** operator is represented as ||, and the logical **XOR** operator is represented as ^. Starting with JDK 8, these operators are applied to two booleans and are wrapped in three static methods—Boolean.logicalAnd(), Boolean.logicalOr(), and Boolean.logicalXor():

int s = 10;  
int m = 21;  
  
// if (s > m && m < 50) { } else { }  
if (Boolean.logicalAnd(s > m, m < 50)) {} else {}  
   
// if (s > m || m < 50) { } else { }  
if (Boolean.logicalOr(s > m, m < 50)) {} else {}  
  
// if (s > m ^ m < 50) { } else { }  
if (Boolean.logicalXor(s > m, m < 50)) {} else {}Copy

Using a combination of these methods is also possible:

if (Boolean.logicalAnd(  
 Boolean.logicalOr(s > m, m < 50),  
 Boolean.logicalOr(s <= m, m > 50))) {} else {}Copy

33. Converting BigInteger into a primitive type

The BigInteger class is a very handy tool for representing immutable arbitrary-precision integers.

This class also contains methods (originating from java.lang.Number) that are useful for converting BigInteger into a primitive type such as byte, long, or double. However, these methods can produce unexpected results and confusion. For example, let's assume that we have BigInteger that wraps Long.MAX\_VALUE:

BigInteger nr = BigInteger.valueOf(Long.MAX\_VALUE);Copy

Let's convert this BigInteger into a primitive long via the BigInteger.longValue() method:

long nrLong = nr.longValue();Copy

So far, everything has worked as expected since the Long.MAX\_VALUE is 9,223,372,036,854,775,807 and the nrLong primitive variable has exactly this value.

Now, let's try to convert this BigInteger class into a primitive int value via the BigInteger.intValue() method:

int nrInt = nr.intValue();Copy

This time, the nrInt primitive variable will have a value of -1 (the same result will produce shortValue() and byteValue()). Conforming to the documentation, if the value of BigInteger is too big to fit in the specified primitive type, only the low-order *n* bits are returned (*n* depends on the specified primitive type). But if the code is not aware of this statement, then it will push values as -1 in further computations, which will lead to confusion.

However, starting with JDK 8, a new set of methods was added. These methods are dedicated to identifying the information that's lost during the conversion from BigInteger into the specified primitive type. If a piece of lost information is detected, ArithmeticException will be thrown. This way, the code signals that the conversion has encountered some issues and prevents this unpleasant situation.

These methods are longValueExact(), intValueExact(), shortValueExact(), and byteValueExact():

long nrExactLong = nr.longValueExact(); // works as expected  
int nrExactInt = nr.intValueExact(); // throws ArithmeticExceptionCopy

Notice that intValueExact() did not return -1 as intValue(). This time, the lost information that was caused by the attempt of converting the largest long value into int was signaled via an exception of the ArithmeticException type.

34. Converting long into int

Converting a long value into an int value seems like an easy job. For example, a potential solution can rely on casting the following:

long nr = Integer.MAX\_VALUE;  
int intNrCast = (int) nr;Copy

Alternatively, it can rely on Long.intValue(), as follows:

int intNrValue = Long.valueOf(nrLong).intValue();Copy

Both approaches work just fine. Now, let's suppose we have the following long value:

long nrMaxLong = Long.MAX\_VALUE;Copy

This time, both approaches will return -1. In order to avoid such results, it is advisable to rely on JDK 8, that is, Math.toIntExact(). This method gets an argument of the long type and tries to convert it into int. If the obtained value overflows int, then this method will throw ArithmeticException:

// throws ArithmeticException  
int intNrMaxExact = Math.toIntExact(nrMaxLong); Copy

Behind the scenes, toIntExact() relies on the ((int)value != value) condition.

35. Computing the floor of a division and modulus

Let's assume that we have the following division:

double z = (double)222/14;Copy

This will initialize z with the result of this division, that is, 15.85, but our problem requests the floor of this division, which is 15 (this is the largest integer value that is less than or equal to the algebraic quotient). A solution to obtain this desired result will consist of applying Math.floor(15.85), which is 15.

However, 222 and 14 are integers, and so this preceding division is written as follows:

int z = 222/14;Copy

This time, z will be equal to 15, which is exactly the expected result (the / operator returns the integer closest to zero). There is no need to apply Math.floor(z). Moreover, if the divisor is 0, then 222/0 will throw ArithmeticException.

The conclusion so far is that the floor of a division for two integers that have the same sign (both are positive or negative) can be obtained via the / operator.

Okay, so far, so good, but let's assume that we have the following two integers (opposite signs; the dividend is negative and the divisor is positive, and vice versa):

double z = (double) -222/14;Copy

This time, z will be equal to -15.85. Again, by applying Math.floor(z), the result will be -16, which is correct (this is the largest integer value that is less than or equal to the algebraic quotient).

Let's go over the same problem again with int:

int z = -222/14;Copy

This time, z will be equal to -15. This is incorrect and Math.floor(z) will not help us in this case since Math.floor(-15) is -15. So, this is a problem that should be considered.

From JDK 8 onward, all of these cases have been covered and exposed via the Math.floorDiv() method. This method takes two integers representing the dividend and the divisor as arguments and returns the largest (closest to positive infinity) int value that is less than or equal to the algebraic quotient:

int x = -222;  
int y = 14;  
  
// x is the dividend, y is the divisor  
int z = Math.floorDiv(x, y); // -16Copy

The Math.floorDiv() method comes in three flavors: floorDiv(int x, int y), floorDiv(long x, int y), and floorDiv(long x, long y).

After Math.floorDiv(), JDK 8 came with Math.floorMod(), which returns the floor modulus of the given arguments. This is computed as the result of x - (floorDiv(x, y) \* y), and so it will return the same result as the % operator for arguments with the same sign and a different result for arguments that don't have the same sign.

Rounding up the result of dividing two positive integers (a/b) can be accomplished quickly as follows:

long result = (a + b - 1) / b;Copy

The following is one example of this (we have *4 / 3 = 1.33* and we want 2):

long result = (4 + 3 - 1) / 3; // 2Copy

The following is another example of this (we have *17 / 7 = 2.42* and we want 3):

long result = (17 + 7 - 1) / 7; // 3Copy

If the integers are not positive, then we can rely on Math.ceil():

long result = (long) Math.ceil((double) a/b);Copy

36. Next floating-point value

Having an integer value such as 10 makes it very easy for us to obtain the next integer-point value, such as 10+1 (in the direction of positive infinity) or 10-1 (in the direction of negative infinity). Trying to achieve the same thing for float or double is not that easy as it is for integers.

Starting with JDK 6, the Math class has been enriched with the nextAfter() method. This method takes two arguments—the initial number (float or double) and the direction (Float/Double.NEGATIVE/POSITIVE\_INFINITY)—and returns the next floating-point value. Here, it is a flavor of this method to return the next-floating point adjacent to 0.1 in the direction of negative infinity:

float f = 0.1f;  
  
// 0.099999994  
float nextf = Math.nextAfter(f, Float.NEGATIVE\_INFINITY);Copy

Starting with JDK 8, the Math class has been enriched with two methods that act as shortcuts for nextAfter() and are faster. These methods are nextDown() and nextUp():

float f = 0.1f;  
  
float nextdownf = Math.nextDown(f); // 0.099999994  
float nextupf = Math.nextUp(f); // 0.10000001  
  
double d = 0.1d;  
  
double nextdownd = Math.nextDown(d); // 0.09999999999999999  
double nextupd = Math.nextUp(d); // 0.10000000000000002Copy

Therefore, nextAfter() in the direction of negative infinity is available via Math.nextDown() and nextAfter(), while in the direction of positive infinity, this is available via Math.nextUp().

37. Multiplying two large int/long values and operation overflow

Let's dive into the solution starting from the \* operator, as shown in the following example:

int x = 10;  
int y = 5;  
int z = x \* y; // 50Copy

This is a very simple approach and works fine for most of the computations that involve int, long, float, and double as well.

Now, let's apply this operator to the following two large numbers (multiply 2,147,483,647 with itself):

int x = Integer.MAX\_VALUE;  
int y = Integer.MAX\_VALUE;  
int z = x \* y; // 1Copy

This time, z will be equal to 1, which is not the expected result, that is, 4,611,686,014,132,420,609. Changing only the z type from int to long will not help. However, changing the types of x and y from int to long will:

long x = Integer.MAX\_VALUE;  
long y = Integer.MAX\_VALUE;  
long z = x \* y; // 4611686014132420609Copy

But the problem will reappear if we have Long.MAX\_VALUE instead of Integer.MAX\_VALUE:

long x = Long.MAX\_VALUE;  
long y = Long.MAX\_VALUE;  
long z = x \* y; // 1Copy

So, computations that overflow the domain and rely on the \* operator will end up in misleading results.

Instead of using these results in further computations, it is better to be informed on time when an overflow operation occurred. JDK 8 comes with the Math.multiplyExact() method. This method tries to multiply two integers. If the result overflows, int will just throw ArithmeticException:

int x = Integer.MAX\_VALUE;  
int y = Integer.MAX\_VALUE;  
int z = Math.multiplyExact(x, y); // throw ArithmeticExceptionCopy

In JDK 8, Math.muliplyExact(int x, int y) returns int and Math.muliplyExact(long x, long y) returns long. In JDK 9, Math.muliplyExact(long, int y) returning long was added as well.

JDK 9 comes with Math.multiplyFull(int x, int y) returning long value. This method is very useful for obtaining the exact mathematical product of two integers as long, as follows:

int x = Integer.MAX\_VALUE;  
int y = Integer.MAX\_VALUE;  
long z = Math.multiplyFull(x, y); // 4611686014132420609Copy

Just for the record, JDK 9 also comes with a method named Math.muliptlyHigh(long x, long y) returning a long. The long value returned by this method represents the most significant 64 bits of the 128-bit product of two 64-bit factors:

long x = Long.MAX\_VALUE;  
long y = Long.MAX\_VALUE;  
// 9223372036854775807 \* 9223372036854775807 = 4611686018427387903  
long z = Math.multiplyHigh(x, y);Copy

In a functional style context, a potential solution will rely on the BinaryOperator functional interface, as follows (simply define the operation of the two operands of the same type):

int x = Integer.MAX\_VALUE;  
int y = Integer.MAX\_VALUE;  
BinaryOperator<Integer> operator = Math::multiplyExact;  
int z = operator.apply(x, y); // throw ArithmeticExceptionCopy

For working with a large number, also focus on the BigInteger (immutable arbitrary-precision integers) and BigDecimal (immutable, arbitrary-precision signed decimal numbers) classes.

38. Fused Multiply Add

The mathematical computation *(a \* b) + c* is heavily exploited in matrix multiplications, which are frequently used in **High-Performance Computing** (**HPC**), AI applications, machine learning, deep learning, neural networks, and so on.

The simplest way to implement this computation relies directly on the \* and + operators, as follows:

double x = 49.29d;  
double y = -28.58d;  
double z = 33.63d;  
double q = (x \* y) + z;Copy

The main problem of this implementation consists of low accuracy and performance caused by two rounding errors (one for the multiply operation and one for the addition operation).

But thanks to Intel AVX's instructions for performing SIMD operations and to JDK 9, which added the Math.fma() method, this computation can be boosted. By relying on Math.fma(), the rounding is done only once using the round to nearest even rounding mode:

double fma = Math.fma(x, y, z);Copy

Notice that this improvement is available for modern Intel processors, so it is not enough to just have JDK 9 in place.

39. Compact number formatting

Starting with JDK 12, a new class for compact number formatting was added. This class is named java.text.CompactNumberFormat. The main goal of this class is to extend the existing Java number formatting API with support for locale and compaction.

A number can be formatted into a short style (for example, *1000* becomes *1K*) or into a long style (for example, *1000* becomes *1 thousand*). These two styles were grouped in the Style enum as SHORT and LONG.

Besides the CompactNumberFormat constructor, CompactNumberFormat can be created via two static methods that are added to the NumberFormat class:

* The first is a compact number format for the default locale with NumberFormat.Style.SHORT:

public static NumberFormat getCompactNumberInstance()Copy

* The second is a compact number format for the specified locale with NumberFormat.Style:

public static NumberFormat getCompactNumberInstance​(  
 Locale locale, NumberFormat.Style formatStyle)Copy

Let's take a close look at formatting and parsing.

Formatting

By default, a number is formatted using RoundingMode.HALF\_EVEN. However, we can explicitly set the rounding mode via NumberFormat.setRoundingMode().

Trying to condense this information into a utility class named NumberFormatters can be achieved as follows:

public static String forLocale(Locale locale, double number) {  
  
 return format(locale, Style.SHORT, null, number);  
}  
  
public static String forLocaleStyle(  
 Locale locale, Style style, double number) {  
  
 return format(locale, style, null, number);  
}  
  
public static String forLocaleStyleRound(  
 Locale locale, Style style, RoundingMode mode, double number) {  
  
 return format(locale, style, mode, number);  
}  
  
private static String format(  
 Locale locale, Style style, RoundingMode mode, double number) {  
  
 if (locale == null || style == null) {  
 return String.valueOf(number); // or use a default format  
 }  
  
 NumberFormat nf = NumberFormat.getCompactNumberInstance(locale,  
 style);  
  
 if (mode != null) {  
 nf.setRoundingMode(mode);  
 }  
  
 return nf.format(number);  
}Copy

Now, let's format the numbers *1000*, *1000000*, and *1000000000* with the US locale, SHORT style, and default rounding mode:

// 1K  
NumberFormatters.forLocaleStyle(Locale.US, Style.SHORT, 1\_000);  
  
// 1M  
NumberFormatters.forLocaleStyle(Locale.US, Style.SHORT, 1\_000\_000);

// 1B  
NumberFormatters.forLocaleStyle(Locale.US, Style.SHORT,   
 1\_000\_000\_000);Copy

We can do the same with the LONG style:

// 1thousand  
NumberFormatters.forLocaleStyle(Locale.US, Style.LONG, 1\_000);  
  
// 1million  
NumberFormatters.forLocaleStyle(Locale.US, Style.LONG, 1\_000\_000);  
  
// 1billion  
NumberFormatters.forLocaleStyle(Locale.US, Style.LONG, 1\_000\_000\_000);Copy

We can also use the ITALIAN locale and SHORT style:

// 1.000  
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.SHORT,   
 1\_000);  
  
// 1 Mln  
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.SHORT,   
 1\_000\_000);  
  
// 1 Mld  
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.SHORT,   
 1\_000\_000\_000);Copy

Finally, we can also use the ITALIAN locale and LONG style:

// 1 mille  
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.LONG,   
 1\_000);  
  
// 1 milione  
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.LONG,   
 1\_000\_000);  
  
// 1 miliardo  
NumberFormatters.forLocaleStyle(Locale.ITALIAN, Style.LONG,   
 1\_000\_000\_000);Copy

Now, let's suppose that we have two numbers: *1200* and *1600*.

From the rounding mode's perspective, they will be rounded to *1000* and *2000*, respectively. The default rounding mode, HALF\_EVEN, will round *1200* to *1000* and *1600* to *2000*. But if we want *1200* to become *2000* and *1600* to become *1000*, then we need to explicitly set up the rounding mode as follows:

// 2000 (2 thousand)  
NumberFormatters.forLocaleStyleRound(  
 Locale.US, Style.LONG, RoundingMode.UP, 1\_200);  
  
// 1000 (1 thousand)  
NumberFormatters.forLocaleStyleRound(  
 Locale.US, Style.LONG, RoundingMode.DOWN, 1\_600);Copy

Parsing

Parsing is the reverse process of formatting. We have a given string and try to parse it as a number. This can be accomplished via the NumberFormat.parse() method. By default, parsing doesn't take advantage of grouping (for example, without grouping, *5,50 K* is parsed as *5*; with grouping, *5,50 K* is parsed as *550000*).

If we condense this information into a set of helper methods, then we obtain the following output:

public static Number parseLocale(Locale locale, String number)   
 throws ParseException {  
  
 return parse(locale, Style.SHORT, false, number);  
}  
  
public static Number parseLocaleStyle(  
 Locale locale, Style style, String number) throws ParseException {  
  
 return parse(locale, style, false, number);  
}  
  
public static Number parseLocaleStyleRound(  
 Locale locale, Style style, boolean grouping, String number)  
 throws ParseException {  
  
 return parse(locale, style, grouping, number);  
}  
  
private static Number parse(  
 Locale locale, Style style, boolean grouping, String number)  
 throws ParseException {  
  
 if (locale == null || style == null || number == null) {  
 throw new IllegalArgumentException(  
 "Locale/style/number cannot be null");  
 }  
  
 NumberFormat nf = NumberFormat.getCompactNumberInstance(locale,   
 style);  
 nf.setGroupingUsed(grouping);  
  
 return nf.parse(number);  
}Copy

Let's parse *5K* and *5 thousand* into *5000* without explicit grouping:

// 5000  
NumberFormatters.parseLocaleStyle(Locale.US, Style.SHORT, "5K");  
  
// 5000  
NumberFormatters.parseLocaleStyle(Locale.US, Style.LONG, "5 thousand");Copy

Now, let's parse *5,50K* and *5,50 thousand* to *550000* with explicit grouping:

// 550000  
NumberFormatters.parseLocaleStyleRound(  
 Locale.US, Style.SHORT, true, "5,50K");  
  
// 550000  
NumberFormatters.parseLocaleStyleRound(  
 Locale.US, Style.LONG, true, "5,50 thousand");Copy

More tuning can be obtained via the setCurrency​(), setParseIntegerOnly(), setMaximumIntegerDigits(), setMinimumIntegerDigits(), setMinimumFractionDigits(), and setMaximumFractionDigits() methods.

Summary

This chapter collected a bunch of the most common problems that involve strings and numbers. Obviously, there are tons of such problems, and trying to cover all of them is way beyond any book's scope. However, knowing how to solve the problems presented in this chapter provides you with a solid base for solving many other related problems by yourself.

Download the applications from this chapter to view the results and additional details.

Objects, Immutability, and Switch Expressions

This chapter includes 18 problems that involve objects, immutability, and switch expressions. The chapter starts with several problems about dealing with null references. It continues with problems regarding checking indexes, equals() and hashCode(), and immutability (for example, writing immutable classes and passing/returning mutable objects from immutable classes). The last part of the chapter deals with cloning objects and the JDK 12 switch expressions. By the end of this chapter, you will have a fundamental knowledge of objects and immutability. Moreover, you will know how to deal with the new switch expressions. These are valuable and non-optional bits of knowledge in any Java developer's arsenal.

Problems

Use the following problems to test your object, immutability, and switch expression programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Checking null references in functional style and** **imperative code**:Write a program that performs the null checks on the given references in a functional style and imperative code.
2. **Checking null references and throwing a customized** NullPointerException error:Write a program that performs the null checks on the given references and throws NullPointerException with custom messages.
3. **Checking null references and throwing the specified exception (example,** IllegalArgumentException**)**:Write a program that performs the null checks on the given references and throws the specified exception.
4. **Checking null references and returning non-null default references**:Write a program that performs the null checks on the given reference, and if it is non-null, then return it; otherwise, return a non-null default reference.
5. **Checking the index in the range from 0 to length**: Write a program that checks whether the given index is between 0 (inclusive) and the given length (exclusive). If the given index is out of the [0, *given length*] range, then throw IndexOutOfBoundsException.
6. **Checking the subrange in the range from 0 to length**:Write a program that checks whether the given subrange [*given start*, *given end*] is within the bounds of the range from [0, *given length*]. If the given subrange is not in the [0, *given length*] range, then throw IndexOutOfBoundsException.
7. **equals() and** hashCode()**:**Explain and exemplify how equals() and hashCode() methods work in Java.
8. **Immutable objects in a nutshell:**Explain and exemplify what is an immutable object in Java.
9. **Immutable string:**Explain why the String class is immutable.
10. **Writing an immutable class:**Write a program that represents an immutable class.
11. **Passing/returning mutable objects to/from an immutable class:**Write a program that passes and returns a mutable object to/from an immutable class.
12. **Writing an immutable class via the Builder pattern:**Write a program that represents an implementation of the Builder pattern in an immutable class.
13. **Avoiding bad data in immutable objects:**Write a program that prevents *bad data* in immutable objects.
14. **Cloning objects:**Write a program that exemplifies shallow and deep cloning techniques.
15. **Overriding toString():**Explain and exemplify practices for overriding toString().
16. **switch expressions:**Provide a brief overview of the switch expressions in JDK 12.
17. **Multiple case labels:**Write a snippet of code for exemplifying the JDK 12 switch with multiple case labels.
18. **Statement blocks:**Write a snippet of code for exemplifying the JDK 12 switch with case labels that point to a curly-braced block.

Solutions

The following sections describe solutions to each of the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations shown here include only the most interesting and important details needed to solve the problems. Download the example solutions to see additional details and to experiment with the programs at <https://github.com/PacktPublishing/Java-Coding-Problems>.

40. Checking null references in functional style and imperative code

Independent of functional style or imperative code, checking null references is a common and recommended technique used for mitigating the occurrence of famous NullPointerException exception. This kind of checking is heavily exploited for method arguments to ensure that the passing references will not cause NullPointerException or unexpected behavior.

For example, passing List<Integer> to a method may require at least two null checks. First, the method should ensure that the list reference itself is not null. Second, depending on how the list is used, the method should ensure that the list does not contain null objects:

List<Integer> numbers   
 = Arrays.asList(1, 2, null, 4, null, 16, 7, null);Copy

This list is passed to the following method:

public static List<Integer> evenIntegers(List<Integer> integers) {  
  
 if (**integers == null**) {  
 return Collections.EMPTY\_LIST;  
 }  
  
 List<Integer> evens = new ArrayList<>();  
 for (Integer nr: integers) {  
 if (**nr != null** && nr % 2 == 0) {  
 evens.add(nr);  
 }  
 }  
  
 return evens;  
}Copy

Notice that the preceding code uses the classical checks relying on the == and != operators (integers==null, nr !=null). Starting with JDK 8, the java.util.Objects class contains two methods that wrap the null checks based on these two operators: object == null was wrapped in Objects.isNull(), and object != null was wrapped in Objects.nonNull().

Based on these methods, the preceding code can be rewritten as follows:

public static List<Integer> evenIntegers(List<Integer> integers) {  
  
 if (**Objects.isNull(integers)**) {  
 return Collections.EMPTY\_LIST;  
 }  
  
 List<Integer> evens = new ArrayList<>();  
  
 for (Integer nr: integers) {  
 if (**Objects.nonNull(nr)** && nr % 2 == 0) {  
 evens.add(nr);  
 }  
 }  
  
 return evens;  
}Copy

Now, the code is somehow more expressive, but this is not the main usage of these two methods. Actually, these two methods have been added for another purpose (conforming to API notes)—to be used as predicates in the Java 8 functional style code. In functional style code, the null checks can be accomplished as in the following examples:

public static int sumIntegers(List<Integer> integers) {  
  
 if (integers == null) {  
 throw new IllegalArgumentException("List cannot be null");  
 }  
  
 return integers.stream()  
 .filter(**i -> i != null**)  
 .mapToInt(Integer::intValue).sum();  
}  
  
public static boolean integersContainsNulls(List<Integer> integers) {  
  
 if (integers == null) {  
 return false;  
 }  
  
 return integers.stream()  
 .anyMatch(**i -> i == null**);  
}Copy

It is quite obvious that i -> i != null and i -> i == null are not expressed in the same style with the surrounding code. Let's replace these snippets of code with Objects.nonNull() and Objects.isNull():

public static int sumIntegers(List<Integer> integers) {  
  
 if (integers == null) {  
 throw new IllegalArgumentException("List cannot be null");  
 }  
  
 return integers.stream()  
 .filter(**Objects::nonNull**)  
 .mapToInt(Integer::intValue).sum();  
}  
  
public static boolean integersContainsNulls(List<Integer> integers) {  
  
 if (integers == null) {  
 return false;  
 }  
  
 return integers.stream()  
 .anyMatch(**Objects::isNull**);  
}Copy

Or, we can use the Objects.nonNull() and Objects.isNull() methods for arguments as well:

public static int sumIntegers(List<Integer> integers) {  
  
 if (**Objects.isNull(integers)**) {  
 throw new IllegalArgumentException("List cannot be null");  
 }  
  
 return integers.stream()  
 .filter(**Objects::nonNull**)  
 .mapToInt(Integer::intValue).sum();  
}  
  
public static boolean integersContainsNulls(List<Integer> integers) {  
  
 if (**Objects.isNull(integers)**) {  
 return false;  
 }  
  
 return integers.stream()  
 .anyMatch(**Objects::isNull**);  
}Copy

Awesome! So, by way of conclusion, the functional style code should rely on these two methods whenever the null checks are needed, while in the imperative code, it is a matter of preference.

41. Checking null references and throwing customized NullPointerException

Checking null references and throwing NullPointerException with customized messages can be accomplished using the following code (this code does these four times, twice in the constructor and twice in the assignDriver() method):

public class Car {  
  
 private final String name;  
 private final Color color;  
  
 public Car(String name, Color color) {  
  
 if (name == null) {  
 throw new NullPointerException("Car name cannot be null");  
 }  
  
 if (color == null) {  
 throw new NullPointerException("Car color cannot be null");  
 }  
  
 this.name = name;  
 this.color = color;  
 }  
  
 public void assignDriver(String license, Point location) {  
  
 if (license == null) {  
 throw new NullPointerException("License cannot be null");  
 }  
  
 if (location == null) {  
 throw new NullPointerException("Location cannot be null");  
 }  
 }  
}Copy

So, this code solves the problem by combining the == operator and manual instantiation of the NullPointerException class. Starting with JDK 7, this combination of code was hidden in a static method named Objects.requireNonNull(). Via this method, the preceding code can be rewritten in an expressive manner:

public class Car {  
  
 private final String name;  
 private final Color color;  
  
 public Car(String name, Color color) {  
  
 this.name = Objects.requireNonNull(name, "Car name cannot be   
 null");  
 this.color = Objects.requireNonNull(color, "Car color cannot be   
 null");  
 }  
  
 public void assignDriver(String license, Point location) {  
  
 Objects.requireNonNull(license, "License cannot be null");  
 Objects.requireNonNull(location, "Location cannot be null");  
 }  
}Copy

So, if the specified reference is null, then Objects.requireNonNull() will throw a NullPointerException with the message provided. Otherwise, it returns the checked reference.

In constructors, there is a typical approach to throw NullPointerException when the references provided are null. But in methods (for example, assignDriver()), this is a controversial approach. Some developers will prefer to return an inoffensive result or to throw IllegalArgumentException. The next problem, checking null references and throwing the specified exception (for example, IllegalArgumentException), addresses the IllegalArgumentException approach.

In JDK 7, there are the two Objects.requireNonNull() methods, the one used previously, and another one that throws NullPointerException with a default message, as in the following example:

this.name = Objects.requireNonNull(name);Copy

Starting with JDK 8, there is one more Objects.requireNonNull(). This one wraps the custom message of NullPointerException in Supplier. This means that the message creation is postponed until the given reference is null (this means that using the + operator for concatenating parts of the message is no longer an issue).

Here is an example:

this.name = Objects.requireNonNull(name, ()   
 -> "Car name cannot be null ... Consider one from " + carsList);Copy

If this reference is not null, then the message is not created.

42. Checking null references and throwing the specified exception

Of course, one solution entails relying directly on the == operator as follows:

if (name == null) {  
 throw new IllegalArgumentException("Name cannot be null");  
}Copy

This problem cannot be solved via the methods of java.util.Objects since there is no requireNonNullElseThrow() method. Throwing IllegalArgumentException or another specified exception may require a set of methods, as shown in following screenshot:

Texto

Descripción generada automáticamente

Let's focus on the requireNonNullElseThrowIAE() methods. These two methods throw IllegalArgumentException with a custom message specified as String or as Supplier (to avoid creation until null is evaluated to true):

public static <T> T requireNonNullElseThrowIAE(  
 T obj, String message) {  
  
 if (obj == null) {  
 throw new IllegalArgumentException(message);  
 }  
  
 return obj;  
}  
  
public static <T> T requireNonNullElseThrowIAE(T obj,  
 Supplier<String> messageSupplier) {  
  
 if (obj == null) {  
 throw new IllegalArgumentException(messageSupplier == null   
 ? null : messageSupplier.get());  
 }  
  
 return obj;  
}Copy

So, throwing IllegalArgumentException can be done via these two methods. But they are not enough. For example, the code may need to throw IllegalStateException, UnsupportedOperationException, and so on. For such cases, the following methods are preferable:

public static <T, X extends Throwable> T requireNonNullElseThrow(  
 T obj, X exception) throws X {  
  
 if (obj == null) {  
 throw exception;  
 }  
  
 return obj;  
}  
  
public static <T, X extends Throwable> T requireNotNullElseThrow(  
 T obj, Supplier<<? extends X> exceptionSupplier) throws X {  
  
 if (obj != null) {  
 return obj;  
 } else {  
 throw exceptionSupplier.get();  
 }  
}Copy

Consider adding these methods to a helper class named MyObjects. Call these methods as shown in the following example:

public Car(String name, Color color) {  
  
 this.name = MyObjects.requireNonNullElseThrow(name,  
 new UnsupportedOperationException("Name cannot be set as null"));  
 this.color = MyObjects.requireNotNullElseThrow(color, () ->  
 new UnsupportedOperationException("Color cannot be set as null"));  
}Copy

Furthermore, we can follow these examples to enrich MyObjects with other kinds of exceptions as well.

43. Checking null references and returning non-null default references

A solution to this problem can easily be provided via if-else (or the ternary operator), as in the following example (as a variation, name, and color can be declared as non-final and initialized with the default values at declaration):

public class Car {  
  
 private final String name;  
 private final Color color;  
 public Car(String name, Color color) {  
  
 if (name == null) {  
 this.name = "No name";  
 } else {  
 this.name = name;  
 }  
  
 if (color == null) {  
 this.color = new Color(0, 0, 0);  
 } else {  
 this.color = color;  
 }  
 }  
}Copy

However, starting with JDK 9, the preceding code can be simplified via two methods from the Objects class. These methods are requireNonNullElse() and requireNonNullElseGet(). Both of them take two arguments—the reference to check for nullity, and the non-null default reference to return in case the checked reference is null:

public class Car {  
  
 private final String name;  
 private final Color color;  
  
 public Car(String name, Color color) {  
  
 this.name = Objects.requireNonNullElse(name, "No name");  
 this.color = Objects.requireNonNullElseGet(color,  
 () -> new Color(0, 0, 0));  
 }  
}Copy

In the preceding example, these methods are used in a constructor, but they can be used in methods as well.

44. Checking the index in the range from 0 to length

To begin with, let's have a simple scenario to highlight this problem. This scenario may materialize in the following simple class:

public class Function {  
  
 private final int x;  
  
 public Function(int x) {  
  
 this.x = x;  
 }  
  
 public int xMinusY(int y) {  
  
 return x - y;  
 }  
  
 public static int oneMinusY(int y) {  
  
 return 1 - y;  
 }  
}Copy

Notice that the preceding snippet of code doesn't assume any range restrictions over x and y. Now, let's impose the following ranges (this is very common with mathematical functions):

* x must be between 0 (inclusive) and 11 (exclusive), so x belongs to [0, 11].
* In the xMinusY() method, y must be between 0 (inclusive) and x (exclusive), so y belongs to [0, x].
* In the oneMinusY() method, y must be between 0 (inclusive) and 16 (exclusive), so y belongs to [0, 16).

These ranges can be imposed in code via the if statements, as follows:

public class Function {  
  
 private static final int X\_UPPER\_BOUND = 11;  
 private static final int Y\_UPPER\_BOUND = 16;  
 private final int x;  
  
 public Function(int x) {  
  
 if (x < 0 || x >= X\_UPPER\_BOUND) {  
 throw new IndexOutOfBoundsException("...");   
 }  
  
 this.x = x;  
 }  
  
 public int xMinusY(int y) {  
  
 if (y < 0 || y >= x) {  
 throw new IndexOutOfBoundsException("...");  
 }  
  
 return x - y;  
 }  
  
 public static int oneMinusY(int y) {  
  
 if (y < 0 || y >= Y\_UPPER\_BOUND) {  
 throw new IndexOutOfBoundsException("...");  
 }  
  
 return 1 - y;  
 }  
}Copy

Consider replacing IndexOutOfBoundsException with a more meaningful exception (for example, extend IndexOutOfBoundsException and create a custom exception of type, RangeOutOfBoundsException).

Starting with JDK 9, the code can be rewritten to use the Objects.checkIndex() method. This method verifies whether the given index is in the range [0, *length*] and returns the given index in this range or throws IndexOutOfBoundsException:

public class Function {  
  
 private static final int X\_UPPER\_BOUND = 11;  
 private static final int Y\_UPPER\_BOUND = 16;  
 private final int x;  
  
 public Function(int x) {  
  
 this.x = Objects.checkIndex(x, X\_UPPER\_BOUND);  
 }  
  
 public int xMinusY(int y) {  
  
 Objects.checkIndex(y, x);  
  
 return x - y;  
 }  
  
 public static int oneMinusY(int y) {  
  
 Objects.checkIndex(y, Y\_UPPER\_BOUND);  
  
 return 1 - y;  
 }  
}Copy

For example, calling oneMinusY(), as shown in the next code snippet, will result in IndexOutOfBoundsException since y can take values between [0, 16):

int result = Function.oneMinusY(20);Copy

Now, let's go further and check the subrange in a range from 0 to the given length.

45. Checking the subrange in the range from 0 to length

Let's follow the same flow from the previous problem. So, this time, the Function class will look as follows:

public class Function {  
  
 private final int n;  
  
 public Function(int n) {  
  
 this.n = n;  
 }  
  
 public int yMinusX(int x, int y) {  
  
 return y - x;  
 }  
}Copy

Notice that the preceding snippet of code doesn't assume any range restrictions over x, y, and n. Now, let's impose the following ranges:

* n must be between 0 (inclusive) and 101 (exclusive), so n belongs to [0, 101].
* In the yMinusX() method, the range bounded by x and y, [x, y] must be a subrange of [0, n].

These ranges can be imposed in code via the if statements as follows:

public class Function {  
  
 private static final int N\_UPPER\_BOUND = 101;  
 private final int n;  
  
 public Function(int n) {  
  
 if (n < 0 || n >= N\_UPPER\_BOUND) {  
 throw new IndexOutOfBoundsException("...");  
 }  
  
 this.n = n;  
 }  
  
 public int yMinusX(int x, int y) {  
  
 if (x < 0 || x > y || y >= n) {  
 throw new IndexOutOfBoundsException("...");  
 }  
  
 return y - x;  
 }  
}Copy

Based on the previous problem, the condition for n can be replaced with Objects.checkIndex(). Moreover, the JDK 9 Objects class comes with a method named checkFromToIndex(int start, int end, int length) that checks whether the given subrange [*given start*, *given end*] is within the bounds of the range from [0, *given length*]. So, this method can be applied to the yMinusX() method to check that the range bounded by x and y, [x, y) is a subrange of [0, n]:

public class Function {  
  
 private static final int N\_UPPER\_BOUND = 101;  
 private final int n;  
  
 public Function(int n) {  
  
 this.n = Objects.checkIndex(n, N\_UPPER\_BOUND);  
 }  
  
 public int yMinusX(int x, int y) {  
  
 Objects.checkFromToIndex(x, y, n);  
 return y - x;  
 }  
}Copy

For example, the following test will lead to IndexOutOfBoundsException since x is greater than y:

Function f = new Function(50);  
int r = f.yMinusX(30, 20);Copy

Beside this method, Objects come with another method named checkFromIndexSize(int start, int size, int length). This method checks that the subrange [*given start*, *given start + given size*] is in the range [0, *given length*].

46. equals() and hashCode()

The equals() and hashCode() methods are defined in java.lang.Object. Since Object is the superclass of all Java objects, these two methods are available for all objects. Their main goal is to provide an easy, efficient, and robust solution for comparing objects, and to determine whether they are equal. Without these methods and their contracts, the solution relies on the big and cumbersome if statements meant to compare each field of an object.

When these methods are not overridden, Java will use their default implementations. Unfortunately, the default implementation is not really serving the goal of determining whether two objects have the same value. By default, equals() checks *identity*. In other words, it considers that two objects are equal if, and only if, they are represented by the same memory address (same object references), while hashCode() returns an integer representation of the object memory address. This is a native function known as the *identity* *hash code.*

For example, let's assume the following class:

public class Player {  
  
 private int id;  
 private String name;  
  
 public Player(int id, String name) {  
  
 this.id = id;  
 this.name = name;  
 }  
}Copy

Then, let's create two instances of this class containing the same information, and let's compare them for equality:

Player p1 = new Player(1, "Rafael Nadal");  
Player p2 = new Player(1, "Rafael Nadal");  
  
System.out.println(p1.equals(p2)); // false  
System.out.println("p1 hash code: " + p1.hashCode()); // 1809787067  
System.out.println("p2 hash code: " + p2.hashCode()); // 157627094Copy

Do not use the == operator for testing the equality of objects (avoid if(p1 == p2)). The == operator compares whether the references of two objects are pointing to the same object, whereas equals() compares object values (as humans, this is what we care about).  
  
As a rule of thumb, if two variables hold the same reference, they are *identical,* but if they reference the same value, they are *equal*. What *the same value* means is defined by equals().

For us, p1 and p2 are equal, but notice that equals() has returned false (the p1 and p2 instances have exactly the same field values, but they are stored at different memory addresses). This means that relying on the default implementation of equals() is not acceptable. The solution is to override this method, and for this it is important to be aware of the equals() contract that imposes the following statements:

* **Reflexivity**: An object is equal to itself, which means that p1.equals(p1) must return true.
* **Symmetry**: p1.equals(p2) must return the same result (true/false) as p2.equals(p1).
* **Transitive**: If p1.equals(p2) and p2.equals(p3), then also p1.equals(p3).
* **Consistent**: Two equal objects must remain equal all the time unless one of them is changed.
* **Null returns false**: All objects must be unequal to null.

So, in order to respect this contract, the equals() method of the Player class can be overridden as follows:

@Override  
public boolean equals(Object obj) {  
  
 if (this == obj) {  
 return true;  
 }  
  
 if (obj == null) {  
 return false;  
 }  
  
 if (getClass() != obj.getClass()) {  
 return false;  
 }  
  
 final Player other = (Player) obj;  
  
 if (this.id != other.id) {  
 return false;  
 }  
  
 if (!Objects.equals(this.name, other.name)) {  
 return false;  
 }  
  
 return true;  
}Copy

Now, let's perform the equality test again (this time, p1 is equal to p2):

System.out.println(p1.equals(p2)); // trueCopy

OK, so far so good! Now, let's add these two Player instances to a collection. For example, let's add them to a HashSet (a Java collection that doesn't allow duplicates):

Set<Player> players = new HashSet<>();  
players.add(p1);  
players.add(p2);Copy

Let's check the size of this HashSet and whether it contains p1:

System.out.println("p1 hash code: " + p1.hashCode()); // 1809787067  
System.out.println("p2 hash code: " + p2.hashCode()); // 157627094  
System.out.println("Set size: " + players.size()); // 2  
System.out.println("Set contains Rafael Nadal: "  
 + players.contains(new Player(1, "Rafael Nadal"))); // falseCopy

Conforming to the preceding implementation of equals(), p1, and p2 are equal; therefore, the HashSet size should be 1, not 2. Moreover, it should contain Rafael Nadal. So, what happened?

Well, the general answer resides in how Java was created. It is easy to intuit that equals() is not a fast method; therefore, lookups will face performance penalties when a significant number of equality comparisons are needed. For example, this adds a serious drawback in the case of lookups by specific values in collections (for example, HashSet, HashMap, and HashTable), since it may require a large number of equality comparisons.

Based on this statement, Java tried to reduce equality comparisons by adding *buckets*. A bucket is a hash-based container that groups equal objects. This means that equal objects should return the same hash code, while unequal objects should return different hash codes (if two unequal objects have the same hash code, then this is a *hash collision,* and the objects will go in the same bucket). So, Java compares the hash codes, and only if these are the same for two different object references (not for the same object references) does it proceed further and call equals(). Basically, this accelerates the lookups in collections.

But what happened in our case? Let's see it step by step:

* When p1 is created, Java will assign to it a hash code based on the p1 memory address.
* When p1 is added to Set, Java will link a new bucket to the p1 hash code.
* When p2 is created, Java will assign to it a hash code based on the p2 memory address.
* When p2 is added to Set, Java will link a new bucket to the p2 hash code (when this happens, it looks like HashSet is not working as expected and it allows duplicates).
* When players.contains(new Player(1, "Rafael Nadal")) is executed, a new player, p3, is created with a new hash code based on the p3 memory address.
* So, in the frame of contains(), testing p1 and p3, respectively, p2 and p3 , for equality involves checking their hash codes, and since the p1 hash code is different from the p3 hash code, and the p2 hash code is different from the p3 hash code, the comparisons stop without evaluating equals() and this means that HashSet doesn't contain the object (p3)

In order to get back on track, the code must override the hashCode() method as well. The hashCode() contract imposes the following:

* Two equal objects conforming to equals() must return the same hash code.
* Two objects with the same hash code are not mandatory equals.
* As long as the object remains unchanged, hashCode() must return the same value.

As a rule of thumb, in order to respect the equals() and hashCode() contracts, follow two golden rules:

* When equals() is overridden, hashCode() must be overridden as well, and vice versa.
* Use the same identifying attributes for both methods in the same order.

For the Player class, hashCode() can be overridden as follows:

@Override  
public int hashCode() {  
  
 int hash = 7;  
 hash = 79 \* hash + this.id;  
 hash = 79 \* hash + Objects.hashCode(this.name);  
  
 return hash;  
}Copy

Now, let's execute another test (this time, it works as expected):

System.out.println("p1 hash code: " + p1.hashCode()); // -322171805  
System.out.println("p2 hash code: " + p2.hashCode()); // -322171805  
System.out.println("Set size: " + players.size()); // 1  
System.out.println("Set contains Rafael Nadal: "  
 + players.contains(new Player(1, "Rafael Nadal"))); // trueCopy

Now, let's enumerate some of the common mistakes of working with equals() and hashCode():

* You override equals() and forget to override hashCode() or vice versa (override both or none).
* You use the == operator instead of equals() for comparing object values.
* In equals(), you omit one or more of the following:
  + Start by adding the *self-check* (if (this == obj)...).
  + Since no instance should be equal to null, continue by adding *null-check* (if(obj == null)...).
  + Ensure that the instance is what we are expecting (use getClass() or instanceof).
  + Finally, after these corner-cases, add field comparisons.
* You violate equals() symmetry via inheritance. Assume a class A and a class B extending A and adding a new field. The B class overrides the equals() implementation inherited from A, and this implementation is added to the new field. Relying on instanceof will reveal that b.equals(a) will return false (as expected), but a.equals(b) will return true (not expected), so therefore symmetry is broken. Relying on *slice comparison* will not work since this breaks transitivity and reflexivity. Fixing the problem means relying on getClass() instead of instanceof (via getClass(), instances of the type and its subtypes cannot be equal), or better relying on composition instead of inheritance as in the application bundled to this book (P46\_ViolateEqualsViaSymmetry).
* You return a constant from hashCode() instead of a unique hash code per object.

Since JDK 7, the Objects class has come with several helpers for dealing with object equality and hash codes, as follows:

* Objects.equals(Object a, Object b): Tests whether the a object is equal to the b object.
* Objects.deepEquals(Object a, Object b): Useful for testing whether two objects are equal (if they are arrays, the test is performed via Arrays.deepEquals()).
* Objects.hash(Object ... values): Generates a hash code for a sequence of input values.

Ensure that equals() and hashCode() respect the Java SE contracts via the EqualsVerifier library (<https://mvnrepository.com/artifact/nl.jqno.equalsverifier/equalsverifier>).  
  
Rely on the Lombok library to generate hashCode() and equals() from the fields of your object (<https://projectlombok.org/>). But pay attention to the special case of combining Lombok with JPA entities.

47. Immutable objects in a nutshell

An immutable object is an object that cannot be changed (its state is fixed) once it is created.

In Java, the following applies:

* Primitive types are immutable.
* The famous Java String class is immutable (other classes are immutable as well, for example, Pattern, and LocalDate)
* Arrays are not immutable.
* Collections can be mutable, unmodifiable, or immutable.

An unmodifiable collection is not automatically immutable. It depends on which objects are stored in the collection. If the stored objects are mutable, then the collection is mutable and unmodifiable. But if the stored objects are immutable, then the collection is effectively immutable.

Immutable objects are useful in concurrent (multithread) applications and streams. Since immutable objects cannot be changed, they are impassible to concurrency issues and they don't risk being corrupted or inconsistent.

One of the main concerns of using immutable objects is related to the penalties of creating new objects, instead of managing the state of a mutable object. But keep in mind that immutable objects take advantage of special treatment during garbage collection. Moreover, they are not prone to concurrency issues and eliminate the code needed for managing the state of the mutable objects. The code necessary to manage the state of mutable objects is prone to be slower than the creation of new objects.

Looking at the following problems will allow us to dive deeper into object immutability in Java.

48. Immutable string

Every programming language has a way of representing strings. As primitive types, strings are part of the predefined types, and they are used in almost every type of Java application.

In Java, strings are not represented by a primitive type like int, long, and float. They are represented by a reference type named String. Almost any Java application uses strings, for example, the main() method of a Java application gets as an argument an array of the String type.

The notoriety of String and its wide range of applications means we should know it in detail. Besides knowing how to declare and manipulate strings (for example, reverse, and capitalize) developers should understand why this class was designed in a special or different way. More precisely, why is String immutable? Or maybe this question has a better resonance formulated like this—what are the pros and cons of String being immutable?

Pros of string immutability

Let's take a look at some of the pros of string immutability in the next section.

String constant pool or cached pool

One of the reasons in favor of string immutability is represented by the **string constant pool** (**SCP**) or cached pool. In order to understand this statement, let's dive a little bit into how the String class works internally.

The SCP is a special area in memory (not the normal heap memory) used for the storage of string literals. Let's assume the following three String variables:

String x = "book";  
String y = "book";  
String z = "book";Copy

How many String objects have been created? It is tempting to say three, but actually Java creates only one String object with the "book" value. The idea is that everything between quotes is considered as a string literal, and Java stores string literals in this special area of memory called the SCP, by following an algorithm like this (this algorithm is known as **string interning**):

* When a string literal is created (for example, String x = "book"), Java inspects the SCP to see whether this string literal exists.
* If the string literal is not found in the SCP, then a new string object for the string literal is created in the SCP and the corresponding variable, x, will point to it.
* If the string literal is found in the SCP (for example, String y = "book", String z = "book"), then the new variable will point to the String object (basically, all variables that have the same value will point to the same String object):

Diagrama

Descripción generada automáticamente con confianza media

But x should be "cook" and not "book", so let's replace "b" with "c"—x = x.replace("b", "c");.

While x should be "cook", y and z should remain unchanged. This behavior is provided by immutability. Java will create a new object and will perform the change on it as follows:

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

So, string immutability permits the caching of string literals, which allows applications to use a large number of string literals with a minimum impact on the heap memory and garbage collector. In a mutable context, a modification of a string literal may lead to corrupted variables.

Do not create a string as String x = new String("book"). This is not a string literal; this is a String instance (built via a constructor) that will go in the normal memory heap instead of the SCP. A string created in the normal heap memory can point to the SCP by explicitly calling the String.intern() method as x.intern().

Security

Another benefit of string immutability is its security aspect. Commonly, a lot of sensitive information (usernames, passwords, URLs, ports, databases, socket connections, parameters, properties, and so on) are represented and passed around as strings. By having this information immutable, the code becomes secure to a wide range of security threats (for example, modifying the references accidentally or deliberately).

Thread safety

Imagine an application using thousands of mutable String objects and dealing with thread-safety code. Fortunately, in this case, our imagined scenario will not become a reality, thanks to immutability. Any immutable object is thread-safe by its nature. This means that strings can be shared and manipulated by multiple threads, with no risk of corruption and inconsistency.

Hash code caching

The e*quals() and hashCode()* section discussed equals() and hashCode(). Hash codes should be calculated every time they are involved in hashing specific activities (for example, searching an element in a collection). Since String is immutable, every string has an immutable hash code that can be cached and reused as it cannot be changed after string creation. This means that hash codes of strings can be used from the cache instead of recalculating them at each usage. For example, HashMap hashes its keys for different operations (for example, put(), get()), and if these keys are of the String type, then hash codes will be reused from the cache instead of recalculating them.

Class loading

A typical approach for loading a class in memory relies on calling the Class.forName(String className) method. Notice the String argument representing the class name. Thanks to string immutability, the class name cannot be changed during the loading process. However, if String is mutable, then imagine loading class A (for example, Class.forName("A")), and, during the loading process, its name will get changed to BadA. Now, the BadA objects can do bad things!

Cons of string immutability

Let's take a look at some of the cons of string immutability in the next section.

String cannot be extended

An immutable class should be declared final to avoid extensibility. However, developers need to extend the String class in order to add more features, and this limitation can be considered a drawback of immutability.

Nevertheless, developers can write utility classes (for example, Apache Commons Lang, StringUtils, Spring Framework, StringUtils, Guava, and strings) to provide extra features and simply pass strings as arguments to the methods of these classes.

Sensitive data in memory for a long time

Sensitive data in strings (for example, passwords) may reside in memory (in SCP) for a long time. Being a cache, the SCP takes advantage of special treatment from the garbage collector. More precisely, the SCP is not visited by the garbage collector with the same frequency (cycles) as other memory zones. As a consequence of this special treatment, sensitive data is kept in the SCP for a long time, and can be prone to unwanted usages.

In order to avoid this potential drawback, it is advisable to store sensitive data (for example, passwords) in char[] instead of String.

OutOfMemoryError

The SCP is a small memory zone in comparison with others and can be filled pretty quickly. Storing too many string literals in the SCP will lead to OutOfMemoryError.

Is String completely immutable?

Well, behind the scenes, String uses private final char[] to store each character of the string. By using the Java Reflection API, in JDK 8, the following code will modify this char[] (the same code in JDK 11 will throw java.lang.ClassCastException):

String user = "guest";  
System.out.println("User is of type: " + user);  
  
Class<String> type = String.class;  
Field field = type.getDeclaredField("value");  
field.setAccessible(true);  
  
char[] chars = (char[]) field.get(user);  
  
chars[0] = 'a';  
chars[1] = 'd';  
chars[2] = 'm';  
chars[3] = 'i';  
chars[4] = 'n';  
  
System.out.println("User is of type: " + user);Copy

So, in JDK 8, String is *effectively* immutable, but not *completely*.

49. Writing an immutable class

An immutable class must respect several requirements, such as the following:

* The class should be marked as final to suppress extensibility (other classes cannot extend this class; therefore, they cannot override methods)
* All fields should be declared private and final (they are not visible in other classes, and they are initialized only once in the constructor of this class)
* The class should contain a parameterized public constructor (or a private constructor and factory methods for creating instances) that initializes the fields
* The class should provide getters for fields
* The class should not expose setters

For example, the following Point class is immutable since it successfully passes the preceding checklist:

public final class Point {  
  
 private final double x;  
 private final double y;  
  
 public Point(double x, double y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public double getX() {  
 return x;  
 }  
  
 public double getY() {  
 return y;  
 }  
}Copy

If the immutable class should manipulate mutable objects, consider the following problems.

50. Passing/returning mutable objects to/from an immutable class

Passing mutable objects to an immutable class can break down immutability. Let's consider the following mutable class:

public class Radius {  
  
 private int start;  
 private int end;  
  
 public int getStart() {  
 return start;  
 }  
  
 public void setStart(int start) {  
 this.start = start;  
 }  
  
 public int getEnd() {  
 return end;  
 }  
  
 public void setEnd(int end) {  
 this.end = end;  
 }  
}Copy

Then, let's pass an instance of this class to an immutable class named, Point. At first glance, the Point class can be written as follows:

public final class Point {  
  
 private final double x;  
 private final double y;  
 private final Radius radius;  
  
 public Point(double x, double y, Radius radius) {  
 this.x = x;  
 this.y = y;  
 this.radius = radius;  
 }  
  
 public double getX() {  
 return x;  
 }  
  
 public double getY() {  
 return y;  
 }  
  
 public Radius getRadius() {  
 return radius;  
 }  
}Copy

Is this class still immutable? The answer is—no. The Point class is not immutable anymore because its state can be changed as in the following example:

Radius r = new Radius();  
r.setStart(0);  
r.setEnd(120);  
  
Point p = new Point(1.23, 4.12, r);  
  
System.out.println("Radius start: " + p.getRadius().getStart()); // 0  
r.setStart(5);  
System.out.println("Radius start: " + p.getRadius().getStart()); // 5Copy

Notice that calling p.getRadius().getStart() returned two different results; therefore, the state of p has been changed, so Point is no longer immutable. A solution to this problem is cloning the Radius object and storing the clone as the field of Point:

public final class Point {  
  
 private final double x;  
 private final double y;  
 private final Radius radius;  
  
 public Point(double x, double y, Radius radius) {  
 this.x = x;  
 this.y = y;  
  
 Radius clone = new Radius();  
 clone.setStart(radius.getStart());  
 clone.setEnd(radius.getEnd());  
  
 this.radius = clone;  
 }  
  
 public double getX() {  
 return x;  
 }  
  
 public double getY() {  
 return y;  
 }  
  
 public Radius getRadius() {  
 return radius;  
 }  
}Copy

This time, the Point class immutability level has increased (calling r.setStart(5) will not affect the radius field since this field is a clone of r). But the Point class is not completely immutable because there is one more problem to solve—returning mutable objects from an immutable class can break down immutability. Check the following code that breaks down the immutability of Point:

Radius r = new Radius();  
r.setStart(0);  
r.setEnd(120);  
  
Point p = new Point(1.23, 4.12, r);  
  
System.out.println("Radius start: " + p.getRadius().getStart()); // 0  
p.getRadius().setStart(5);  
System.out.println("Radius start: " + p.getRadius().getStart()); // 5Copy

Again, calling p.getRadius().getStart() returned two different results; therefore, the state of p has been changed. The solution consists of modifying the getRadius() method to return a clone of the radius field, as follows:

...  
public Radius getRadius() {  
 Radius clone = new Radius();  
 clone.setStart(this.radius.getStart());  
 clone.setEnd(this.radius.getEnd());  
  
 return clone;  
 }  
...Copy

Now, the Point class is immutable again. Problem solved!

Before choosing the cloning technique/tool, in certain cases, it is advisable to take your time and analyze/learn different possibilities available in Java and third-party libraries (for example, check the *Cloning objects* section in this chapter). For shallow copies, the preceding technique can be the proper choice, but for deep copies, the code may need to rely on different approaches such as copy constructor, the Cloneable interface, or external libraries (for example, Apache Commons Lang ObjectUtils, JSON serialization with Gson or Jackson, or any others).

51. Writing an immutable class via the Builder pattern

When a class (immutable or mutable) has too many fields, it requires a constructor with many arguments. When some of those fields are required and others are optional, this class will need several constructors to cover all the possible combinations. This becomes cumbersome for the developer and for the user of the class. This is where the Builder pattern comes to the rescue.

According to the **Gang of Four** (**GoF**)—*the Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.*

The Builder pattern can be implemented as a separate class or as an inner static class. Let's focus on the second case. The User class has three required fields (nickname, password, and created) and three optional fields (email, firstname, and lastname).

Now, an immutable User class relying on the Builder pattern will appear as follows:

public final class User {  
  
 private final String nickname;  
 private final String password;  
 private final String firstname;  
 private final String lastname;  
 private final String email;  
 private final Date created;  
  
 private User(UserBuilder builder) {  
 this.nickname = builder.nickname;  
 this.password = builder.password;  
 this.created = builder.created;  
 this.firstname = builder.firstname;  
 this.lastname = builder.lastname;  
 this.email = builder.email;  
 }  
  
 public static UserBuilder getBuilder(  
 String nickname, String password) {  
 return new User.UserBuilder(nickname, password);  
 }  
  
 public static final class UserBuilder {  
  
 private final String nickname;  
 private final String password;  
 private final Date created;  
 private String email;  
 private String firstname;  
 private String lastname;  
  
 public UserBuilder(String nickname, String password) {  
 this.nickname = nickname;  
 this.password = password;  
 this.created = new Date();  
 }  
  
 public UserBuilder firstName(String firstname) {  
 this.firstname = firstname;  
 return this;  
 }  
  
 public UserBuilder lastName(String lastname) {  
 this.lastname = lastname;  
 return this;  
 }  
  
 public UserBuilder email(String email) {  
 this.email = email;  
 return this;  
 }  
  
 public User build() {  
 return new User(this);  
 }  
 }  
  
 public String getNickname() {  
 return nickname;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public String getFirstname() {  
 return firstname;  
 }  
  
 public String getLastname() {  
 return lastname;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public Date getCreated() {  
 return new Date(created.getTime());  
 }  
}Copy

Here are some usage examples:

import static modern.challenge.User.getBuilder;  
...  
// user with nickname and password  
User user1 = getBuilder("marin21", "hjju9887h").build();  
  
// user with nickname, password and email  
User user2 = getBuilder("ionk", "44fef22")  
 .email("ion@gmail.com")  
 .build();  
  
// user with nickname, password, email, firstname and lastname  
User user3 = getBuilder("monika", "klooi0988")  
 .email("monika@gmail.com")  
 .firstName("Monika")  
 .lastName("Ghuenter")  
 .build();Copy

52. Avoiding bad data in immutable objects

*Bad data* is any data that has a negative impact on the immutable object (for example, corrupted data). Most probably, this data comes from user inputs or from external data sources that are not under our direct control. In such cases, bad data can hit the immutable object, and the worst part is that there is no fix for it. An immutable object cannot be changed after creation; therefore, bad data will live happily as long as the object lives.

The solution to this problem is to validate all data that enters in an immutable object against a comprehensive set of constraints.

There are different ways of performing validation, from custom validation to built-in solutions. Validation can be performed outside or inside the immutable object class, depending on the application design. For example, if the immutable object is built via the Builder pattern, then the validation can be performed in the builder class.

JSR 380 is a specification of the Java API for bean validation (Java SE/EE) that can be used for validation via annotations. Hibernate Validator is the reference implementation of the validation API, and it can be easily provided as a Maven dependency in the pom.xml file (check the source code bundled to this book).

Furthermore, we rely on dedicated annotations to provide the needed constraints (for example, @NotNull, @Min, @Max, @Size, and @Email). In the following example, the constraints are added to the builder class as follows:

...  
public static final class UserBuilder {  
  
 @NotNull(message = "cannot be null")  
 @Size(min = 3, max = 20, message = "must be between 3 and 20   
 characters")  
 private final String nickname;  
  
 @NotNull(message = "cannot be null")  
 @Size(min = 6, max = 50, message = "must be between 6 and 50   
 characters")  
 private final String password;  
  
 @Size(min = 3, max = 20, message = "must be between 3 and 20   
 characters")  
 private String firstname;  
  
 @Size(min = 3, max = 20, message = "must be between 3 and 20   
 characters")  
 private String lastname;  
  
 @Email(message = "must be valid")  
 private String email;  
  
 private final Date created;  
  
 public UserBuilder(String nickname, String password) {  
 this.nickname = nickname;  
 this.password = password;  
 this.created = new Date();  
 }  
...Copy

Finally, the validation process is triggered from code via the Validator API (this is needed in Java SE only). If the data that enters the builder class is invalid, then the immutable object is not created (don't call the build() method):

User user;  
Validator validator   
 = Validation.buildDefaultValidatorFactory().getValidator();  
  
User.UserBuilder userBuilder   
 = new User.UserBuilder("monika", "klooi0988")  
 .email("monika@gmail.com")  
 .firstName("Monika").lastName("Gunther");  
  
final Set<ConstraintViolation<User.UserBuilder>> violations   
 = validator.validate(userBuilder);  
if (violations.isEmpty()) {  
 user = userBuilder.build();  
 System.out.println("User successfully created on: "   
 + user.getCreated());  
} else {  
 printConstraintViolations("UserBuilder Violations: ", violations);  
}Copy

This way, the bad data cannot touch an immutable object. If there is no builder class, then the constraints can be added directly at the field level in the immutable object. The preceding solution simply displays the potential violations on the console, but, depending on the situation, the solution may perform different actions (for example, throw specific exceptions).

53. Cloning objects

Cloning objects is not a daily task, but it is important to do it properly. Mainly, cloning objects refers to creating copies of objects. There are two main types of copies—*shallow* copies (copy as little as possible) and *deep* copies (copy everything).

Let's assume the following class:

public class Point {  
  
 private double x;  
 private double y;  
  
 public Point() {}  
 public Point(double x, double y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 // getters and setters  
}Copy

So, we have a point of type (x, y) mapped in a class. Now, let's perform some cloning.

Manual cloning

A quick approach consists of adding a method that copies the current Point to a new Point manually (this is a shallow copy):

public Point clonePoint() {  
 Point point = new Point();  
 point.setX(this.x);  
 point.setY(this.y);  
  
 return point;  
}Copy

The code here is pretty simple. Just create a new instance of Point and populate its fields with the fields of the current Point. The returned Point is a shallow copy (since Point doesn't depend on other objects, a deep copy will be exactly the same) of the current Point:

Point point = new Point(...);  
Point clone = point.clonePoint();Copy

Cloning via clone()

The Object class contains a method named clone(). This method is useful for creating shallow copies (it can be used for deep copies as well). In order to use it, a class should follow the given steps:

* Implement the Cloneable interface (if this interface is not implemented, then CloneNotSupportedException will be thrown).
* Override the clone() method (Object.clone() is protected).
* Call super.clone().

The Cloneable interface doesn't contain any methods. It is just a signal for JVM that this object can be cloned. Once this interface is implemented, the code needs to override the Object.clone() method. This is needed because Object.clone() is protected, and, in order to call it via super, the code needs to override this method. This can be a serious drawback if clone() is added to a child class since all superclasses should define a clone() method in order to avoid the failure of the super.clone() chain invocation.

Moreover, Object.clone() doesn't rely on a constructor invocation, and so the developer cannot control the object construction:

public class Point implements Cloneable {  
  
 private double x;  
 private double y;  
  
 public Point() {}  
  
 public Point(double x, double y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 @Override  
 public Point clone() throws CloneNotSupportedException {  
 return (Point) super.clone();  
 }  
  
 // getters and setters  
}Copy

Creating a clone can be done as follows:

Point point = new Point(...);  
Point clone = point.clone();Copy

Cloning via a constructor

This cloning technique requires you to enrich the class with a constructor that takes a single argument representing an instance of the class that will be used to create the clone.

Let's see it in code:

public class Point {  
  
 private double x;  
 private double y;  
  
 public Point() {}  
  
 public Point(double x, double y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public Point(Point another) {  
 this.x = another.x;  
 this.y = another.y;  
 }  
  
 // getters and setters  
}Copy

Creating a clone can be done as follows:

Point point = new Point(...);  
Point clone = new Point(point);Copy

Cloning via the Cloning library

A deep copy is needed when an object depends on another object. Performing a deep copy means copying the object, including its chain of dependencies. For example, let's assume that Point has a field of the Radius type:

public class Radius {  
  
 private int start;  
 private int end;  
  
 // getters and setters  
}  
  
public class Point {  
  
 private double x;  
 private double y;  
 private Radius radius;  
  
 public Point(double x, double y, Radius radius) {  
 this.x = x;  
 this.y = y;  
 this.radius = radius;  
 }  
  
 // getters and setters  
}Copy

Performing a shallow copy of Point will create a copy of x and y, but will not create a copy of the radius object. This means that modifications that affect the radius object will be reflected in the clone as well. It's time for a deep copy.

A cumbersome solution will involve adapting the shallow copy techniques previously presented to support a deep copy. Fortunately, there are a few solutions that can be applied out of the box, and one of them is the Cloning library (<https://github.com/kostaskougios/cloning>):

import com.rits.cloning.Cloner;  
...  
Point point = new Point(...);  
Cloner cloner = new Cloner();  
Point clone = cloner.deepClone(point);Copy

The code is self-explanatory. Notice that the Cloning library comes with several other goodies as well, as can be seen in the following screenshot:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Cloning via serialization

This technique requires serializable objects (implement java.io.Serializable). Basically, the object is serialized (writeObject()) and deserialized (readObject()) in a new object. A helper method able to accomplish this is listed as follows:

private static <T> T cloneThroughSerialization(T t) {  
  
 try {  
 ByteArrayOutputStream baos = new ByteArrayOutputStream();  
 ObjectOutputStream oos = new ObjectOutputStream(baos);  
 oos.writeObject(t);  
  
 ByteArrayInputStream bais   
 = new ByteArrayInputStream(baos.toByteArray());  
 ObjectInputStream ois = new ObjectInputStream(bais);  
  
 return (T) ois.readObject();  
 } catch (IOException | ClassNotFoundException ex) {  
 // log exception  
 return t;  
 }  
}Copy

So, the object is serialized in ObjectOutputStream and deserialized in ObjectInputStream. Cloning an object via this method can be accomplished as follows:

Point point = new Point(...);  
Point clone = cloneThroughSerialization(point);Copy

A built-in solution based on serialization is provided by Apache Commons Lang, via SerializationUtils. Among its methods, this class provides a method named clone() that can be used as follows:

Point point = new Point(...);  
Point clone = SerializationUtils.clone(point);Copy

Cloning via JSON

Almost any JSON library in Java can serialize any **Plain Old Java Object** (**POJO**) without any extra configuration/mapping required. Having a JSON library in the project (and many projects have) can save us from adding an extra library to provide deep cloning. Mainly, the solution can leverage the existing JSON library to get the same effect.

The following is an example using the Gson library:

private static <T> T cloneThroughJson(T t) {  
  
 Gson gson = new Gson();  
 String json = gson.toJson(t);  
  
 return (T) gson.fromJson(json, t.getClass());  
}  
  
Point point = new Point(...);  
Point clone = cloneThroughJson(point);Copy

In addition to this, there is always the option of writing your own library dedicated to cloning objects.

54. Overriding toString()

The toString() method is defined in java.lang.Object, and the JDK comes with a default implementation of it. This default implementation is automatically used for all objects that are the subject of print(), println(), printf(), debugging during development, logging, informative messages in exceptions, and so on.

Unfortunately, the string representation of an object returned by the default implementation is not very informative. For example, let's consider the following User class:

public class User {  
 private final String nickname;  
 private final String password;  
 private final String firstname;  
 private final String lastname;  
 private final String email;  
 private final Date created;  
  
 // constructor and getters skipped for brevity  
}Copy

Now, let's create an instance of this class, and let's print it on the console:

User user = new User("sparg21", "kkd454ffc",  
 "Leopold", "Mark", "markl@yahoo.com");  
  
System.out.println(user);Copy

The output of this println() method will be something like the following:

Imagen que contiene Diagrama

Descripción generada automáticamente

The solution for avoiding outputs as in the preceding screenshot consists of overriding the toString() method. For example, let's override it to expose the user details, as follows:

@Override  
public String toString() {  
 return "User{" + "nickname=" + nickname + ", password=" + password  
 + ", firstname=" + firstname + ", lastname=" + lastname  
 + ", email=" + email + ", created=" + created + '}';  
}Copy

This time, println() will reveal the following output:

User {  
 nickname = sparg21, password = kkd454ffc,   
 firstname = Leopold, lastname = Mark,   
 email = markl@yahoo.com, created = Fri Feb 22 10: 49: 32 EET 2019  
}Copy

This is much more informative than the previous output.

But, remember that toString() is automatically called for different purposes. For example, logging can be as follows:

logger.log(Level.INFO, "This user rocks: {0}", user);Copy

Here, the user password will hit the log, and this may represent a problem. Exposing log-sensitive data, such as passwords, accounts, and secret IPs, in an application is definitely a bad practice.

Therefore, pay extra attention to carefully selecting the information that goes in toString(), since this information may end up in places where it can be maliciously exploited. In our case, the password should not be part of toString():

@Override  
public String toString() {  
 return "User{" + "nickname=" + nickname  
 + ", firstname=" + firstname + ", lastname=" + lastname  
 + ", email=" + email + ", created=" + created + '}';  
}Copy

Commonly, toString() is a method generated via an IDE. So, pay attention to which fields you select before the IDE generates the code for you.

55. Switch expressions

Before we have a brief overview of the switch expressions introduced in JDK 12, let's see a typical old-school example wrapped in a method:

private static Player createPlayer(PlayerTypes playerType) {  
  
 switch (playerType) {  
  
 case TENNIS:  
 return new TennisPlayer();  
 case FOOTBALL:  
 return new FootballPlayer();  
 case SNOOKER:   
 return new SnookerPlayer();  
 case UNKNOWN:  
 throw new UnknownPlayerException("Player type is unknown");  
 default:  
 throw new IllegalArgumentException(  
 "Invalid player type: " + playerType);  
  
 }  
}Copy

If we forget about default, then the code will not compile.

Obviously, the preceding example is acceptable. In the worst-case scenario, we can add a spurious variable (for example, player), some cluttering break statements, and get no complaints if default is missing. So, the following code is an old-school, extremely ugly switch:

private static Player createPlayerSwitch(PlayerTypes playerType) {  
  
 Player player = null;  
  
 switch (playerType) {  
 case TENNIS:  
 player = new TennisPlayer();  
 break;  
 case FOOTBALL:  
 player = new FootballPlayer();  
 break;  
 case SNOOKER:  
 player = new SnookerPlayer();  
 break;  
 case UNKNOWN:  
 throw new UnknownPlayerException(  
 "Player type is unknown");  
 default:  
 throw new IllegalArgumentException(  
 "Invalid player type: " + playerType);  
 }  
  
 return player;  
}Copy

If we forget about default, then there will be no complaints from the compiler side. In this case, a missing default case may result in a null player.

However, since JDK 12, we have been able to rely on the switch expressions. Before JDK 12, switch was a statement, a construct meant to control the flow (for example, as an if statement) without representing the result. On the other hand, an expression is evaluated to a result. Therefore, a switch expression can have a result.

The preceding switch expression can be written in the style of JDK 12 as follows:

private static Player createPlayer(PlayerTypes playerType) {  
  
 return switch (playerType) {  
 case TENNIS ->  
 new TennisPlayer();  
 case FOOTBALL ->  
 new FootballPlayer();  
 case SNOOKER ->  
 new SnookerPlayer();  
 case UNKNOWN ->  
 throw new UnknownPlayerException(  
 "Player type is unknown");  
 // default is not mandatory  
 default ->  
 throw new IllegalArgumentException(  
 "Invalid player type: " + playerType);  
 };  
}Copy

This time, default is not mandatory. We can skip it.

The JDK 12 switch is smart enough to signal if switch doesn't cover all possible input values. This is very useful in the case of Java enum values. The JDK 12 switch can detect whether all the enum values are covered, and doesn't force a useless default if they aren't. For example, if we remove default and add a new entry to PlayerTypes enum (for example, GOLF), then the compiler will signal it via a message, as in the following screenshot (this is from NetBeans):

Imagen que contiene Texto

Descripción generada automáticamente

Notice that between the label and execution, we've replaced the colon with an arrow (the lambda-style syntax). The main role of this arrow is to prevent fall-through, which means that only the block of code from its right will be executed. There is no need to use break.

Do not conclude that the arrow turns the switch statement into a switch expression. A switch expression can be used with a colon and break as well, as follows:

private static Player createPlayer(PlayerTypes playerType) {  
  
 return switch (playerType) {  
 case TENNIS:  
 break new TennisPlayer();  
 case FOOTBALL:  
 break new FootballPlayer();  
 case SNOOKER:  
 break new SnookerPlayer();  
 case UNKNOWN:  
 throw new UnknownPlayerException(  
 "Player type is unknown");  
 // default is not mandatory  
 default:  
 throw new IllegalArgumentException(  
 "Invalid player type: " + playerType);  
 };  
}Copy

Our example posts switch over enum, but the JDK 12 switch can also be used over int, Integer, short, Short, byte, Byte, char, Character, and String.  
  
Notice that JDK 12 brings the switch expressions as a preview feature. This means that it is prone to changes in the next few releases, and it needs to be unlocked via the --enable-preview command-line option at compiling and runtime.

56. Multiple case labels

Before JDK 12, a switch statement allowed a single label per case. Starting with the switch expressions, a case can have multiple labels separated by a comma. Check out the following method that exemplifies multiple case labels:

private static SportType   
 fetchSportTypeByPlayerType(PlayerTypes playerType) {  
  
 return switch (playerType) {  
 case TENNIS, GOLF, SNOOKER ->  
 new Individual();  
 case FOOTBALL, VOLLEY ->   
 new Team();   
 };  
}Copy

So, if we pass to this method TENNIS, GOLF, or SNOOKER, it will return an instance of the Individual class. If we pass FOOTBALL or VOLLEY, it will return an instance of the Team class.

57. Statement blocks

A label's arrow can point to a single statement (as in the examples from the previous two problems) or to a curly-braced block. This is pretty similar to the lambda blocks. Check out the following solution:

private static Player createPlayer(PlayerTypes playerType) {  
 return switch (playerType) {  
 case TENNIS -> {  
 System.out.println("Creating a TennisPlayer ...");  
 break new TennisPlayer();  
 }  
 case FOOTBALL -> {  
 System.out.println("Creating a FootballPlayer ...");  
 break new FootballPlayer();  
 }  
 case SNOOKER -> {  
 System.out.println("Creating a SnookerPlayer ...");  
 break new SnookerPlayer();  
 }  
 default ->  
 throw new IllegalArgumentException(  
 "Invalid player type: " + playerType);  
 };  
}Copy

Notice that we exit from a curly-braced block via break, not return. In other words, while we can return from inside a switch statement, we can't return from within an expression.

Summary

That's all folks! This chapter has introduced you to several problems involving objects, immutability, and the switch expressions. While the problems covering objects and immutability represent fundamental concepts of programming, the problems covering the switch expressions were dedicated to introducing the new JDK 12 features addressing this topic.

Download the applications from this chapter to see the results and to see additional details.

Working with Date and Time

This chapter includes 20 problems that involve date and time. These problems are meant to cover a wide range of topics (converting, formatting, adding, subtracting, defining periods/durations, computing, and so on) via Date, Calendar, LocalDate, LocalTime, LocalDateTime, ZoneDateTime, OffsetDateTime, OffsetTime, Instant, and so on. By the end of this chapter, you will have no problems in shaping date and time, while conforming to your application's needs. The fundamental problems presented in this chapter will be very helpful for obtaining the bigger picture regarding date-time APIs, and will act like the pieces of the puzzle that need to be pieced together in order to resolve complex challenges involving date and time.

Problems

Use the following problems to test your date and time programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Converting a string to date and time**:Write a program that exemplifies conversions between a string and date/time.
2. **Formatting date and time**:Explain the format pattern for date and time.
3. **Getting the current date/time without time/date**:Write a program that extracts the current date without the time or date.
4. LocalDateTime **from** LocalDate **and** LocalTime:Write a program that builds a LocalDateTime from LocalDate object and LocalTime. It combines the date and time in a single LocalDateTime object.
5. **Machine time via an** Instant **class**:Explain and give an example of the Instant API.
6. **Defining a period of time using date-based values (Period) and a duration of time using time-based values (Duration)**:Explain and give an example of the usage of the Period and Duration APIs.
7. **Getting date and time units**: Write a program that extracts the date and time units (for example, extract from date the year, month, minute, and so on) from an object representing a date-time.
8. **Adding and subtracting to/from a date-time**: Write a program that adds (and subtracts) an amount of time (for example, years, days, or minutes) to a date-time object (for example, add an hour to date, subtract 2 days from LocalDateTime, and so on).
9. **Getting all time zones with UTC and GMT**:Write a program that displays all the available time zones with UTC and GMT.
10. **Getting the local date-time in all available time zones**: Write a program that displays the local time in all the available time zones.
11. **Displaying date-time information about a flight**: Write a program that displays information about a scheduled flight time of 15 hours and 30 minutes. More precisely, a flight from Perth, Australia to Bucharest, Europe.
12. **Converting a Unix timestamp to date-time**: Write a program that converts a Unix timestamp to java.util.Date and java.time.LocalDateTime.
13. **Finding the first/last day of the month**: Write a program that finds the first/last day of the month via JDK 8, TemporalAdjusters.
14. **Defining/extracting zone offsets**: Write a program that reveals different techniques for defining and extracting zone offsets.
15. **Converting between** Date **and** Temporal: Write a program that converts between Date and Instant, LocalDate, LocalDateTime, and so on.
16. **Iterating a range of dates**:Write a program that iterates a range of given dates, day by day (with a step of a day).
17. **Calculating age**:Write a program that calculates the age of a person.
18. **Start and end of a day**: Write a program that returns the start and end time of a day.
19. **Difference between two dates:**Write a program that calculates the amount of time, in days, between two dates.
20. **Implementing a chess clock:**Write a program that implements a chess clock.

Solutions

The following sections describe solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations shown here include only the most interesting and important details that are needed to solve the problems. Download the example solutions to see additional details and to experiment with the programs at <https://github.com/PacktPublishing/Java-Coding-Problems>.

58. Converting a string to date and time

Converting or parsing String to date and time can be accomplished via a set of parse() methods. Converting from date and time to String can be accomplished via the toString() or format() methods.

Before JDK 8

Before JDK 8, the typical solution to this problem relies on the main extension of the abstract DateFormat class, named SimpleDateFormat (this is not a thread-safe class). In the code that is bundled to this book, there are several examples of how to use this class.

Starting with JDK 8

Starting with JDK 8, SimpleDateFormat can be replaced with a new class—DateTimeFormatter. This is an immutable (and, therefore, thread-safe) class, and is used for printing and parsing date-time objects. This class supports everything from predefined formatters (represented as constants, as the ISO local date, 2011-12-03, is ISO\_LOCAL\_DATE) to user-defined formatters (relying on a set of symbols for writing custom format patterns).

Moreover, beside the Date class, JDK 8 comes with several new classes, which are dedicated to working with date and time. Some of these classes are shown in the following list (these are also referenced as temporals because they implement the Temporal interface):

* LocalDate (date without a time zone in the ISO-8601 calendar system)
* LocalTime (time without a time zone in the ISO-8601 calendar system)
* LocalDateTime (date-time without a time zone in the ISO-8601 calendar system)
* ZonedDateTime (date-time with a time zone in the ISO-8601 calendar system), and so on
* OffsetDateTime (date-time with an offset from UTC/GMT in the ISO-8601 calendar system)
* OffsetTime (time with an offset from UTC/GMT in the ISO-8601 calendar system)

In order to convert String to LocalDate via a predefined formatter, it should respect the DateTimeFormatter.ISO\_LOCAL\_DATE pattern, for example, 2020-06-01. LocalDate provides a parse() method that can be used as follows:

// 06 is the month, 01 is the day  
LocalDate localDate = LocalDate.parse("2020-06-01");Copy

Similarly, in the case of LocalTime, the string should respect the DateTimeFormatter.ISO\_LOCAL\_TIME pattern; for example, 10:15:30, as shown in the following code snippet:

LocalTime localTime = LocalTime.parse("12:23:44");Copy

In the case of LocalDateTime, the string should respect the DateTimeFormatter.ISO\_LOCAL\_DATE\_TIME pattern; for example, 2020-06-01T11:20:15, as shown in the following code snippet:

LocalDateTime localDateTime   
 = LocalDateTime.parse("2020-06-01T11:20:15");Copy

And, in case of ZonedDateTime, the string must respect the DateTimeFormatter.ISO\_ZONED\_DATE\_TIME pattern; for example, 2020-06-01T10:15:30+09:00[Asia/Tokyo], as shown in the following code snippet:

ZonedDateTime zonedDateTime   
 = ZonedDateTime.parse("2020-06-01T10:15:30+09:00[Asia/Tokyo]");Copy

In the case of OffsetDateTime, the string must respect the DateTimeFormatter.ISO\_OFFSET\_DATE\_TIME pattern; for example, 2007-12-03T10:15:30+01:00, as shown in the following code snippet:

OffsetDateTime offsetDateTime   
 = OffsetDateTime.parse("2007-12-03T10:15:30+01:00");Copy

Finally, in the case of OffsetTime, the string must respect the DateTimeFormatter.ISO\_OFFSET\_TIME pattern; for example, 10:15:30+01:00, as shown in the following code snippet:

OffsetTime offsetTime = OffsetTime.parse("10:15:30+01:00");Copy

If the string doesn't respect any of the predefined formatters, then it is time for a user-defined formatter via a custom format pattern; for example, the string 01.06.2020 represents a date that needs a user-defined formatter, as follows:

DateTimeFormatter dateFormatter   
 = DateTimeFormatter.ofPattern("dd.MM.yyyy");  
LocalDate localDateFormatted   
 = LocalDate.parse("01.06.2020", dateFormatter);Copy

However, a string such as 12|23|44 requires a user-defined formatter as follows:

DateTimeFormatter timeFormatter   
 = DateTimeFormatter.ofPattern("HH|mm|ss");  
LocalTime localTimeFormatted   
 = LocalTime.parse("12|23|44", timeFormatter);Copy

A string such as 01.06.2020, 11:20:15 requires a user-defined formatter as follows:

DateTimeFormatter dateTimeFormatter   
 = DateTimeFormatter.ofPattern("dd.MM.yyyy, HH:mm:ss");  
LocalDateTime localDateTimeFormatted   
 = LocalDateTime.parse("01.06.2020, 11:20:15", dateTimeFormatter);Copy

A string such as 01.06.2020, 11:20:15+09:00 [Asia/Tokyo] requires a user-defined formatter as follows:

DateTimeFormatter zonedDateTimeFormatter   
 = DateTimeFormatter.ofPattern("dd.MM.yyyy, HH:mm:ssXXXXX '['VV']'");  
ZonedDateTime zonedDateTimeFormatted   
 = ZonedDateTime.parse("01.06.2020, 11:20:15+09:00 [Asia/Tokyo]",   
 zonedDateTimeFormatter);Copy

A string such as 2007.12.03, 10:15:30, +01:00 requires a user-defined formatter as follows:

DateTimeFormatter offsetDateTimeFormatter   
 = DateTimeFormatter.ofPattern("yyyy.MM.dd, HH:mm:ss, XXXXX");  
OffsetDateTime offsetDateTimeFormatted   
 = OffsetDateTime.parse("2007.12.03, 10:15:30, +01:00",   
 offsetDateTimeFormatter);Copy

Finally, a string such as 10 15 30 +01:00 requires a user-defined formatter as follows:

DateTimeFormatter offsetTimeFormatter   
 = DateTimeFormatter.ofPattern("HH mm ss XXXXX");  
OffsetTime offsetTimeFormatted   
 = OffsetTime.parse("10 15 30 +01:00", offsetTimeFormatter);Copy

Each ofPattern() method from the previous examples also supports Locale.

Converting from LocalDate, LocalDateTime, or ZonedDateTime to String can be accomplished in at least two ways:

* Rely on the LocalDate, LocalDateTime, or ZonedDateTime.toString() method (automatically or explicitly). Notice that relying on toString() will always print the date via the corresponding predefined formatter:

// 2020-06-01 results in ISO\_LOCAL\_DATE, 2020-06-01  
String localDateAsString = localDate.toString();  
  
// 01.06.2020 results in ISO\_LOCAL\_DATE, 2020-06-01  
String localDateAsString = localDateFormatted.toString();  
  
// 2020-06-01T11:20:15 results   
// in ISO\_LOCAL\_DATE\_TIME, 2020-06-01T11:20:15  
String localDateTimeAsString = localDateTime.toString();  
  
// 01.06.2020, 11:20:15 results in   
// ISO\_LOCAL\_DATE\_TIME, 2020-06-01T11:20:15  
String localDateTimeAsString   
 = localDateTimeFormatted.toString();  
  
// 2020-06-01T10:15:30+09:00[Asia/Tokyo]   
// results in ISO\_ZONED\_DATE\_TIME,  
// 2020-06-01T11:20:15+09:00[Asia/Tokyo]  
String zonedDateTimeAsString = zonedDateTime.toString();  
  
// 01.06.2020, 11:20:15+09:00 [Asia/Tokyo]   
// results in ISO\_ZONED\_DATE\_TIME,  
// 2020-06-01T11:20:15+09:00[Asia/Tokyo]  
String zonedDateTimeAsString   
 = zonedDateTimeFormatted.toString();Copy

* Rely on the DateTimeFormatter.format() method. Notice that relying on DateTimeFormatter.format() will always print the date/time using the specified formatter (by default, the time zone will be null), as follows:

// 01.06.2020  
String localDateAsFormattedString   
 = dateFormatter.format(localDateFormatted);  
  
// 01.06.2020, 11:20:15  
String localDateTimeAsFormattedString   
 = dateTimeFormatter.format(localDateTimeFormatted);  
  
// 01.06.2020, 11:20:15+09:00 [Asia/Tokyo]  
String zonedDateTimeAsFormattedString   
 = zonedDateTimeFormatted.format(zonedDateTimeFormatter);Copy

Adding an explicit time zone into the discussion can be done as follows:

DateTimeFormatter zonedDateTimeFormatter   
 = DateTimeFormatter.ofPattern("dd.MM.yyyy, HH:mm:ssXXXXX '['VV']'")  
 .withZone(ZoneId.of("Europe/Paris"));  
ZonedDateTime zonedDateTimeFormatted   
 = ZonedDateTime.parse("01.06.2020, 11:20:15+09:00 [Asia/Tokyo]",   
 zonedDateTimeFormatter);Copy

This time, the string represents the date/time in the Europe/Paris time zone:

// 01.06.2020, 04:20:15+02:00 [Europe/Paris]  
String zonedDateTimeAsFormattedString   
 = zonedDateTimeFormatted.format(zonedDateTimeFormatter);Copy

59. Formatting date and time

The previous problem contains some flavors of formatting date and time via SimpleDateFormat.format() and DateTimeFormatter.format(). In order to define *format patterns,* the developer must be aware of the format pattern syntax. In other words, the developer must be aware of the set of symbols that are used by the Java date-time API in order to recognize a valid format pattern.

Most of the symbols are common to SimpleDateFormat (before JDK 8) and to DateTimeFormatter (starting with JDK 8). The following table lists the most common symbols—the complete list is available in the JDK documentation:

|  |  |  |  |
| --- | --- | --- | --- |
| **Letter** | **Meaning** | **Presentation** | **Example** |
| y | year | year | 1994; 94 |
| M | month of year | number/text | 7; 07; Jul; July; J |
| W | week of month | number | 4 |
| E | day of week | text | Tue; Tuesday; T |
| d | day of month | number | 15 |
| H | hour of day | number | 22 |
| m | minute of hour | number | 34 |
| s | second of minute | number | 55 |
| S | fraction of second | number | 345 |
| z | time zone name | zone-name | Pacific Standard Time; PST |
| Z | zone offset | zone-offset | -0800 |
| V | time zone id (JDK 8) | zone-id | America/Los\_Angeles; Z; -08:30 |

Some format pattern examples are available in the following table:

|  |  |
| --- | --- |
| **Pattern** | **Example** |
| yyyy-MM-dd | 2019-02-24 |
| MM-dd-yyyy | 02-24-2019 |
| MMM-dd-yyyy | Feb-24-2019 |
| dd-MM-yy | 24-02-19 |
| dd.MM.yyyy | 24.02.2019 |
| yyyy-MM-dd HH:mm:ss | 2019-02-24 11:26:26 |
| yyyy-MM-dd HH:mm:ssSSS | 2019-02-24 11:36:32743 |
| yyyy-MM-dd HH:mm:ssZ | 2019-02-24 11:40:35+0200 |
| yyyy-MM-dd HH:mm:ss z | 2019-02-24 11:45:03 EET |
| E MMM yyyy HH:mm:ss.SSSZ | Sun Feb 2019 11:46:32.393+0200 |
| yyyy-MM-dd HH:mm:ss VV (JDK 8) | 2019-02-24 11:45:41 Europe/Athens |

Before JDK 8, a format pattern can be applied via SimpleDateFormat:

// yyyy-MM-dd  
Date date = new Date();  
SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");  
String stringDate = formatter.format(date);Copy

Starting with JDK 8, a format pattern can be applied via DateTimeFormatter:

* For LocalDate (date without a time zone in the ISO-8601 calendar system):

// yyyy-MM-dd  
LocalDate localDate = LocalDate.now();  
DateTimeFormatter formatterLocalDate   
 = DateTimeFormatter.ofPattern("yyyy-MM-dd");  
String stringLD = formatterLocalDate.format(localDate);  
  
// or shortly  
String stringLD = LocalDate.now()  
 .format(DateTimeFormatter.ofPattern("yyyy-MM-dd"));Copy

* For LocalTime (time without a time zone in the ISO-8601 calendar system):

// HH:mm:ss  
LocalTime localTime = LocalTime.now();  
DateTimeFormatter formatterLocalTime   
 = DateTimeFormatter.ofPattern("HH:mm:ss");  
String stringLT   
 = formatterLocalTime.format(localTime);  
  
// or shortly  
String stringLT = LocalTime.now()  
 .format(DateTimeFormatter.ofPattern("HH:mm:ss"));Copy

* For LocalDateTime (date-time without a time zone in the ISO-8601 calendar system):

// yyyy-MM-dd HH:mm:ss  
LocalDateTime localDateTime = LocalDateTime.now();  
DateTimeFormatter formatterLocalDateTime   
 = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");  
String stringLDT   
 = formatterLocalDateTime.format(localDateTime);  
  
// or shortly  
String stringLDT = LocalDateTime.now()  
 .format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));Copy

* For ZonedDateTime (date-time with a time zone in the ISO-8601 calendar system):

// E MMM yyyy HH:mm:ss.SSSZ  
ZonedDateTime zonedDateTime = ZonedDateTime.now();  
DateTimeFormatter formatterZonedDateTime   
 = DateTimeFormatter.ofPattern("E MMM yyyy HH:mm:ss.SSSZ");  
String stringZDT   
 = formatterZonedDateTime.format(zonedDateTime);  
  
// or shortly  
String stringZDT = ZonedDateTime.now()  
 .format(DateTimeFormatter  
 .ofPattern("E MMM yyyy HH:mm:ss.SSSZ"));Copy

* For OffsetDateTime (date-time with an offset from UTC/GMT in the ISO-8601 calendar system):

// E MMM yyyy HH:mm:ss.SSSZ  
OffsetDateTime offsetDateTime = OffsetDateTime.now();  
DateTimeFormatter formatterOffsetDateTime   
 = DateTimeFormatter.ofPattern("E MMM yyyy HH:mm:ss.SSSZ");  
String odt1 = formatterOffsetDateTime.format(offsetDateTime);  
  
// or shortly  
String odt2 = OffsetDateTime.now()  
 .format(DateTimeFormatter  
 .ofPattern("E MMM yyyy HH:mm:ss.SSSZ"));Copy

* For OffsetTime (time with an offset from UTC/GMT in the ISO-8601 calendar system):

// HH:mm:ss,Z  
OffsetTime offsetTime = OffsetTime.now();  
DateTimeFormatter formatterOffsetTime   
 = DateTimeFormatter.ofPattern("HH:mm:ss,Z");  
String ot1 = formatterOffsetTime.format(offsetTime);  
  
// or shortly  
String ot2 = OffsetTime.now()  
 .format(DateTimeFormatter.ofPattern("HH:mm:ss,Z"));Copy

60. Getting the current date/time without time/date

Before JDK 8, the solution must focus on the java.util.Date class. The code that is bundled to this book contains this solution.

Starting with JDK 8, the date and time can be obtained via the dedicated classes, LocalDate and LocalTime, from the java.time package:

// 2019-02-24  
LocalDate onlyDate = LocalDate.now();  
  
// 12:53:28.812637300  
LocalTime onlyTime = LocalTime.now();Copy

61. LocalDateTime from LocalDate and LocalTime

The LocalDateTime class exposes a series of of() methods that are useful for obtaining a different kind of instance of LocalDateTime. For example, a LocalDateTime class that is obtained from the year, month, day, hour, minute, second, or nanosecond looks like this:

LocalDateTime ldt = LocalDateTime.of​(2020, 4, 1, 12, 33, 21, 675);Copy

So, the preceding code combines date and time as arguments of the of() method. In order to combine date and time as objects, the solution can take advantage of the following of() method:

public static LocalDateTime of​(LocalDate date, LocalTime time)Copy

This results in LocalDate and LocalTime, as in the following:

LocalDate localDate = LocalDate.now(); // 2019-Feb-24  
LocalTime localTime = LocalTime.now(); // 02:08:10 PMCopy

They can be combined in a single object, LocalDateTime, as follows:

LocalDateTime localDateTime = LocalDateTime.of(localDate, localTime);Copy

Formatting LocalDateTime reveals the date and time as follows:

// 2019-Feb-24 02:08:10 PM  
String localDateTimeAsString = localDateTime  
 .format(DateTimeFormatter.ofPattern("yyyy-MMM-dd hh:mm:ss a"));Copy

62. Machine time via an Instant class

JDK 8 comes with a new class, which is named java.time.Instant. Mainly, the Instant class represents an instantaneous point on the timeline, beginning from the first second of January 1, 1970 (the epoch), in the UTC time zone with a resolution of nanoseconds.

A Java 8 Instant class is similar in concept to java.util.Date. Both represent a moment on the timeline in UTC. While Instant has a resolution up to nanoseconds, java.util.Date has a milliseconds resolution.

This class is very handy for generating timestamps of machine time. In order to obtain such a timestamp, simply call the now() method as follows:

// 2019-02-24T15:05:21.781049600Z  
Instant timestamp = Instant.now();Copy

A similar output can be obtained with the following code snippet:

OffsetDateTime now = OffsetDateTime.now(ZoneOffset.UTC);Copy

Alternatively, use this code snippet:

Clock clock = Clock.systemUTC();Copy

Calling Instant.toString() produces an output that follows the ISO-8601 standard for representing date and time.

Converting String to Instant

A string that follows the ISO-8601 standard for representing date and time can be easily converted to Instant via the Instant.parse() method, as in the following example:

// 2019-02-24T14:31:33.197021300Z  
Instant timestampFromString =  
 Instant.parse("2019-02-24T14:31:33.197021300Z");Copy

Adding or subtracting time to/from Instant

For adding time, Instant has a suite of methods. For example, adding 2 hours to the current timestamp can be accomplished as follows:

Instant twoHourLater = Instant.now().plus(2, ChronoUnit.HOURS);Copy

In terms of subtracting time, for example, 10 minutes, use the following code snippet:

Instant tenMinutesEarlier = Instant.now()  
 .minus(10, ChronoUnit.MINUTES);Copy

Beside the plus() method, Instant also contains plusNanos(), plusMillis(), and plusSeconds(). Moreover, beside the minus() method, Instant also contains minusNanos(), minusMillis(), and minusSeconds().

Comparing Instant objects

Comparing two Instant objects can be accomplished via the Instant.isAfter() and Instant.isBefore() methods. For example, let's look at the following two Instant objects:

Instant timestamp1 = Instant.now();  
Instant timestamp2 = timestamp1.plusSeconds(10);Copy

Check whether timestamp1 is after timestamp2:

boolean isAfter = timestamp1.isAfter(timestamp2); // falseCopy

Check whether timestamp1 is before timestamp2:

boolean isBefore = timestamp1.isBefore(timestamp2); // trueCopy

The time difference between two Instant objects can be computed via the Instant.until() method:

// 10 seconds  
long difference = timestamp1.until(timestamp2, ChronoUnit.SECONDS);Copy

Converting between Instant and LocalDateTime, ZonedDateTime, and OffsetDateTime

These common conversions can be accomplished as in the following examples:

* Convert between Instant and LocalDateTime—since LocalDateTime has no idea of time zone, use a zero offset UTC+0:

// 2019-02-24T15:27:13.990103700  
LocalDateTime ldt = LocalDateTime.ofInstant(  
 Instant.now(), ZoneOffset.UTC);  
  
// 2019-02-24T17:27:14.013105Z  
Instant instantLDT = LocalDateTime.now().toInstant(ZoneOffset.UTC);Copy

* Convert between Instant and ZonedDateTime—convert an Instant UTC+0 to a Paris ZonedDateTime UTC+1:

// 2019-02-24T16:34:36.138393100+01:00[Europe/Paris]  
ZonedDateTime zdt = Instant.now().atZone(ZoneId.of("Europe/Paris"));  
  
// 2019-02-24T16:34:36.150393800Z  
Instant instantZDT = LocalDateTime.now()  
 .atZone(ZoneId.of("Europe/Paris")).toInstant();Copy

* Convert between Instant and OffsetDateTime—specify an offset of 2 hours:

// 2019-02-24T17:34:36.151393900+02:00  
OffsetDateTime odt = Instant.now().atOffset(ZoneOffset.of("+02:00"));  
  
// 2019-02-24T15:34:36.153394Z  
Instant instantODT = LocalDateTime.now()  
 .atOffset(ZoneOffset.of("+02:00")).toInstant();Copy

63. Defining a period of time using date-based values and a duration of time using time-based values

JDK 8 comes with two new classes, named java.time.Period and java.time.Duration. Let's take a detailed look at them in the next sections.

Period of time using date-based values

The Period class is meant to represent an amount of time using date-based values (years, months, weeks, and days). This period of time can be obtained in different ways. For example, a period of 120 days can be obtained as follows:

Period fromDays = Period.ofDays(120); // P120DCopy

Next to the ofDays() method, the Period class also has ofMonths(), ofWeeks(), and ofYears().

Or, a period of 2,000 years, 11 months and 24 days can be obtained via the of() method, as follows:

Period periodFromUnits = Period.of(2000, 11, 24); // P2000Y11M24DCopy

Period can also be obtained from LocalDate:

LocalDate localDate = LocalDate.now();  
Period periodFromLocalDate = Period.of(localDate.getYear(),  
 localDate.getMonthValue(), localDate.getDayOfMonth());Copy

Finally, Period can be obtained from a String object that respects the ISO-8601 period formats PnYnMnD and PnW. For example, the P2019Y2M25D string represents 2019 years, 2 months, and 25 days:

Period periodFromString = Period.parse("P2019Y2M25D");Copy

Calling Period.toString() will return the period while also respecting the ISO-8601 period formats, PnYnMnD and PnW (for example, P120D, P2000Y11M24D).]

But, the real power of Period is revealed when it is used to represent a period of time between two dates (for example, LocalDate). The period of time between March 12, 2018 and July 20, 2019 can be represented as follows:

LocalDate startLocalDate = LocalDate.of(2018, 3, 12);  
LocalDate endLocalDate = LocalDate.of(2019, 7, 20);  
Period periodBetween = Period.between(startLocalDate, endLocalDate);Copy

The amount of time in years, months, and days can be obtained via Period.getYears(), Period.getMonths(), and Period.getDays(). For example, the following helper method uses these methods to output the amount of time as a string:

public static String periodToYMD(Period period) {  
  
 StringBuilder sb = new StringBuilder();  
   
 sb.append(period.getYears())  
 .append("y:")  
 .append(period.getMonths())  
 .append("m:")  
 .append(period.getDays())  
 .append("d");  
  
 return sb.toString();  
}Copy

Let's call this method periodBetween (the difference is 1 year, 4 months, and 8 days):

periodToYMD(periodBetween); // 1y:4m:8dCopy

The Period class is also useful when determining whether a particular date is earlier than another date. There is a flag method, named isNegative(). Having an A period and a B period, the result of applying Period.between(A, B) can be negative if B is before A, or positive if A is before B. Taking this logic further, isNegative() returns true if B is before A or false if A is before B, as in our case that follows (basically, this method returns false if years, months, or days is negative):

// returns false, since 12 March 2018 is earlier than 20 July 2019  
periodBetween.isNegative();Copy

Finally, Period can be modified by adding or subtracting a period of time. There are methods such as plusYears(), plusMonths(), plusDays(), minusYears(), minusMonths(), and minusDays(). For example, adding 1 year to periodBetween can be done as follows:

Period periodBetweenPlus1Year = periodBetween.plusYears(1L);Copy

Adding two Period classes can be accomplished via the Period.plus() method, as follows:

Period p1 = Period.ofDays(5);  
Period p2 = Period.ofDays(20);  
Period p1p2 = p1.plus(p2); // P25DCopy

Duration of time using time-based values

The Duration class is meant to represent an amount of time using time-based values (hours, minutes, seconds, or nanoseconds). This duration of time can be obtained in different ways. For example, a duration of 10 hours can be obtained as follows:

Duration fromHours = Duration.ofHours(10); // PT10HCopy

Next to the ofHours() method, the Duration class also has ofDays(), ofMillis(), ofMinutes(), ofSeconds(), and ofNanos().

Alternatively, a duration of 3 minutes can be obtained via the of() method, as follows:

Duration fromMinutes = Duration.of(3, ChronoUnit.MINUTES); // PT3MCopy

Duration can also be obtained from LocalDateTime:

LocalDateTime localDateTime   
 = LocalDateTime.of(2018, 3, 12, 4, 14, 20, 670);  
  
// PT14M  
Duration fromLocalDateTime   
 = Duration.ofMinutes(localDateTime.getMinute());Copy

It can also be obtained from LocalTime:

LocalTime localTime = LocalTime.of(4, 14, 20, 670);  
  
// PT0.00000067S  
Duration fromLocalTime = Duration.ofNanos(localTime.getNano());Copy

Finally, Duration can be obtained from a String object that respects the ISO-8601 duration format PnDTnHnMn.nS, with days considered to be exactly 24 hours. For example, the P2DT3H4M string has 2 days, 3 hours, and 4 minutes:

Duration durationFromString = Duration.parse("P2DT3H4M");Copy

Calling Duration.toString() will return the duration that respects the ISO-8601 duration format, PnDTnHnMn.nS (for example, PT10H, PT3M, or PT51H4M).

But, as in the case of Period, the real power of Duration is revealed when it is used to represent a period of time between two times (for example, Instant). The duration of time between November 3, 2015, 12:11:30, and December 6, 2016, 15:17:10, can be represented as the difference between two Instant classes, as follows:

Instant startInstant = Instant.parse("2015-11-03T12:11:30.00Z");  
Instant endInstant = Instant.parse("2016-12-06T15:17:10.00Z");  
  
// PT10059H5M40S  
Duration durationBetweenInstant   
 = Duration.between(startInstant, endInstant);Copy

In seconds, this difference can be obtained via the Duration.getSeconds() method:

durationBetweenInstant.getSeconds(); // 36212740 secondsCopy

Or, the duration of time between March 12, 2018, 04:14:20.000000670 and July 20, 2019, 06:10:10.000000720, can be represented as the difference between two LocalDateTime objects, as follows:

LocalDateTime startLocalDateTime   
 = LocalDateTime.of(2018, 3, 12, 4, 14, 20, 670);  
LocalDateTime endLocalDateTime   
 = LocalDateTime.of(2019, 7, 20, 6, 10, 10, 720);  
// PT11881H55M50.00000005S, or 42774950 seconds  
Duration durationBetweenLDT   
 = Duration.between(startLocalDateTime, endLocalDateTime);Copy

Finally, the duration of time between 04:14:20.000000670 and 06:10:10.000000720, can be represented as the difference between two LocalTime objects, as follows:

LocalTime startLocalTime = LocalTime.of(4, 14, 20, 670);  
LocalTime endLocalTime = LocalTime.of(6, 10, 10, 720);  
  
// PT1H55M50.00000005S, or 6950 seconds  
Duration durationBetweenLT   
 = Duration.between(startLocalTime, endLocalTime);Copy

In the preceding examples, Duration was expressed in seconds via the Duration.getSeconds() method—this is the number of seconds in the Duration class. However, the Duration class contains a set of methods that are dedicated to expressing Duration in other time units—in days via toDays(), in hours via toHours(), in minutes via toMinutes(), in milliseconds via toMillis(), and in nanoseconds via toNanos().

Converting from one unit of time to another unit of time may result in a remnant. For example, converting from seconds to minutes may result in a remnant of seconds (for example, 65 seconds is 1 minute and 5 seconds (5 seconds is the remnant)). The remnant can be obtained via the following set of methods—the remnant in days via toDaysPart(), the remnant in hours via toHoursPart(), the remnant in minutes via toMinutesPart(), and so on.

Let's assume that the difference should be displayed as days:hours:minutes:seconds:nano (for example, 9d:2h:15m:20s:230n). Joining the forces of the to*Foo()* and to*Foo*Part() methods in a helper method will result in the following code:

public static String durationToDHMSN(Duration duration) {  
  
 StringBuilder sb = new StringBuilder();  
 sb.append(duration.toDays())  
 .append("d:")  
 .append(duration.toHoursPart())  
 .append("h:")  
 .append(duration.toMinutesPart())  
 .append("m:")  
 .append(duration.toSecondsPart())  
 .append("s:")  
 .append(duration.toNanosPart())  
 .append("n");  
  
 return sb.toString();  
}Copy

Let's call this method durationBetweenLDT (the difference is 495 days, 1 hour, 55 minutes, 50 seconds, and 50 nanoseconds):

// 495d:1h:55m:50s:50n  
durationToDHMSN(durationBetweenLDT);Copy

Identical to the Period class, the Duration class has a flag method named isNegative(). This method is useful when determining whether a particular time is earlier than another time. Having duration A and duration B, the result of applying Duration.between(A, B) can be negative if B is before A, or positive if A is before B. Taking the logic further, isNegative() returns true if B is before A, or false if A is before B, as in the following case:

durationBetweenLT.isNegative(); // falseCopy

Finally, Duration can be modified by adding or subtracting a duration of time. There are methods such as plusDays(), plusHours(), plusMinutes(), plusMillis(), plusNanos(), minusDays(), minusHours(), minusMinutes(), minusMillis(), and minusNanos() to perform this. For example, adding 5 hours to durationBetweenLT can be done as follows:

Duration durationBetweenPlus5Hours = durationBetweenLT.plusHours(5);Copy

Adding two Duration classes can be accomplished via the Duration.plus() method, as follows:

Duration d1 = Duration.ofMinutes(20);  
Duration d2 = Duration.ofHours(2);  
  
Duration d1d2 = d1.plus(d2);  
  
System.out.println(d1 + "+" + d2 + "=" + d1d2); // PT2H20MCopy

64. Getting date and time units

For a Date object, the solution may rely on a Calendar instance. The code that is bundled to this book contains this solution.

For JDK 8 classes, Java provides dedicated get*Foo*() methods and a get​(TemporalField field) method. For example, let's assume the following LocalDateTime object:

LocalDateTime ldt = LocalDateTime.now();Copy

Relying on get*Foo*() methods, we get the following code:

int year = ldt.getYear();  
int month = ldt.getMonthValue();  
int day = ldt.getDayOfMonth();  
int hour = ldt.getHour();  
int minute = ldt.getMinute();  
int second = ldt.getSecond();  
int nano = ldt.getNano();Copy

Or, relying on get​(TemporalField field) results in the following:

int yearLDT = ldt.get(ChronoField.YEAR);  
int monthLDT = ldt.get(ChronoField.MONTH\_OF\_YEAR);  
int dayLDT = ldt.get(ChronoField.DAY\_OF\_MONTH);  
int hourLDT = ldt.get(ChronoField.HOUR\_OF\_DAY);  
int minuteLDT = ldt.get(ChronoField.MINUTE\_OF\_HOUR);  
int secondLDT = ldt.get(ChronoField.SECOND\_OF\_MINUTE);  
int nanoLDT = ldt.get(ChronoField.NANO\_OF\_SECOND);Copy

Notice that the months are counted from one, which is January.

For example, a LocalDateTime object of 2019-02-25T12:58:13.109389100 can be cut into date-time units, resulting in the following:

Year: 2019 Month: 2 Day: 25 Hour: 12 Minute: 58 Second: 13 Nano: 109389100Copy

With a little intuition and documentation, it is very easy to adapt this example for LocalDate, LocalTime, ZonedDateTime, and others.

65. Adding and subtracting to/from date-time

The solution to this problem relies on the Java APIs that are dedicated to manipulating date and time. Let's take a look at them in the next sections.

Working with Date

For a Date object, the solution may rely on a Calendar instance. The code that is bundled to this book contains this solution.

Working with LocalDateTime

Jumping to JDK 8, the focus is on LocalDate, LocalTime, LocalDateTime, Instant, and many more. The new Java date-time API comes with methods that are dedicated to adding or subtracting an amount of time. LocalDate, LocalTime, LocalDateTime, ZonedDateTime, OffsetDateTime, Instant, Period, Duration, and many others come with methods such as plus*Foo*() and minus*Foo*(), where *Foo* can be replaced with the unit of time (for example, plusYears(), plusMinutes(), minusHours(), minusSeconds(), and so on).

Let's assume the following LocalDateTime:

// 2019-02-25T14:55:06.651155500  
LocalDateTime ldt = LocalDateTime.now();Copy

Adding 10 minutes is as easy as calling LocalDateTime.plusMinutes(long minutes), while subtracting 10 minutes is as easy as calling LocalDateTime.minusMinutes(long minutes):

LocalDateTime ldtAfterAddingMinutes = ldt.plusMinutes(10);  
LocalDateTime ldtAfterSubtractingMinutes = ldt.minusMinutes(10);Copy

The output will reveal the following dates:

After adding 10 minutes: 2019-02-25T15:05:06.651155500  
After subtracting 10 minutes: 2019-02-25T14:45:06.651155500Copy

Beside the methods dedicated per time unit, these classes also support plus/minus(TemporalAmount amountToAdd) and plus/minus(long amountToAdd, TemporalUnit unit).

Now, let's focus on the Instant class. Besides plus/minusSeconds(), plus/minusMillis(), and plus/minusNanos(), the Instant class also provides a plus/minus(TemporalAmount amountToAdd) method.

In order to exemplify this method, let's assume the following Instant:

// 2019-02-25T12:55:06.654155700Z  
Instant timestamp = Instant.now();Copy

Now, let's add and subtract 5 hours:

Instant timestampAfterAddingHours   
 = timestamp.plus(5, ChronoUnit.HOURS);  
Instant timestampAfterSubtractingHours   
 = timestamp.minus(5, ChronoUnit.HOURS);Copy

The output will reveal the following Instant:

After adding 5 hours: 2019-02-25T17:55:06.654155700Z  
After subtracting 5 hours: 2019-02-25T07:55:06.654155700ZCopy

66. Getting all time zones with UTC and GMT

UTC and GMT are recognized as the standard references for dealing with dates and times. Today, UTC is the preferred way to go, but UTC and GMT should return the same result in most cases.

In order to get all the time zones with UTC and GMT, the solution should focus on the implementation before and after JDK 8. So, let's start with the solution that was useful before JDK 8.

Before JDK 8

The solution needs to extract the available time zone IDs (Africa/Bamako, Europe/Belgrade, and so on). Furthermore, each time zone ID should be used to create a TimeZone object. Finally, the solution needs to extract the offset that was specific to each time zone, taking into account Daylight Saving Time. The code that is bundled to this book contains this solution.

Starting with JDK 8

The new Java date-time API provides new leverages for solving this problem.

At the first step, the available time zones IDs can be obtained via the ZoneId class, as follows:

Set<String> zoneIds = ZoneId.getAvailableZoneIds();Copy

At the second step, each time zone ID should be used to create a ZoneId instance. This can be accomplished via the ZoneId.of(String zoneId) method:

ZoneId zoneid = ZoneId.of(*current\_zone\_Id*);Copy

At the third step, each ZoneId can be used to obtain the time that is specific to the identified zone. This means that a "lab rats" reference date-time is needed. This reference date-time (without a time zone, LocalDateTime.now()) is combined with the given time zone (ZoneId), via LocalDateTime.atZone(), in order to obtain ZoneDateTime (a date-time that is time-zone aware):

LocalDateTime now = LocalDateTime.now();  
ZonedDateTime zdt = now.atZone(ZoneId.of(*zone\_id\_instance*));Copy

The atZone() method matches the date-time as closely as possible, taking into account time zone rules, such as Daylight Saving Time.

At the fourth step, the code can exploit ZonedDateTime in order to extract the UTC offset (for example, for Europe/Bucharest the UTC offset is +02:00):

String utcOffset = zdt.getOffset().getId().replace("Z", "+00:00");Copy

The getId() method returns the normalized zone offset ID. The +00:00 offset is returned as the Z character; therefore the code needs to quickly replace Z with +00:00, in order to align with the rest of the offsets, which respect the format +hh:mm or +hh:mm:ss.

Now, let's join these steps into a helper method:

public static List<String> fetchTimeZones(OffsetType type) {  
  
 List<String> timezones = new ArrayList<>();  
 Set<String> zoneIds = ZoneId.getAvailableZoneIds();  
 LocalDateTime now = LocalDateTime.now();  
  
 zoneIds.forEach((zoneId) -> {  
 timezones.add("(" + type + now.atZone(ZoneId.of(zoneId))  
 .getOffset().getId().replace("Z", "+00:00") + ") " + zoneId);  
 });  
  
 return timezones;  
}Copy

Assuming that this method lives in a DateTimes class, the following code is obtained:

List<String> timezones   
 = DateTimes.fetchTimeZones(DateTimes.OffsetType.GMT);  
Collections.sort(timezones); // optional sort  
timezones.forEach(System.out::println);Copy

In addition, an output snapshot is shown, as follows:

(GMT+00:00) Africa/Abidjan  
(GMT+00:00) Africa/Accra  
(GMT+00:00) Africa/Bamako  
...  
(GMT+11:00) Australia/Tasmania  
(GMT+11:00) Australia/Victoria  
...Copy

67. Getting local date-time in all available time zones

The solution to this problem can be obtained by following these steps:

1. Get the local date-time.
2. Get the available time zones.
3. Before JDK 8, use SimpleDateFormat with the setTimeZone() method.
4. Starting with JDK 8, use ZonedDateTime.

Before JDK 8

Before JDK 8, the quick solution to get the current local date-time was to call the Date empty constructor. Furthermore, use Date to display it in all the available time zones, which can be obtained via the TimeZone class. The code that is bundled to this book contains this solution.

Starting with JDK 8

Starting with JDK 8, a convenient solution to get the current local date-time in the default time zone is to call the ZonedDateTime.now() method:

ZonedDateTime zlt = ZonedDateTime.now();Copy

So, this is the current date in the default time zone. Furthermore, this date should be displayed in all the available time zones that are obtained via the ZoneId class:

Set<String> zoneIds = ZoneId.getAvailableZoneIds();Copy

Finally, the code can loop the zoneIds, and for each zone id, it can call the ZonedDateTime.withZoneSameInstant(ZoneId zone) method. This method returns a copy of this date-time with a different time zone, retaining the instant:

public static List<String> localTimeToAllTimeZones() {  
  
 List<String> result = new ArrayList<>();  
 Set<String> zoneIds = ZoneId.getAvailableZoneIds();  
 DateTimeFormatter formatter   
 = DateTimeFormatter.ofPattern("yyyy-MMM-dd'T'HH:mm:ss a Z");  
 ZonedDateTime zlt = ZonedDateTime.now();  
  
 zoneIds.forEach((zoneId) -> {  
 result.add(zlt.format(formatter) + " in " + zoneId + " is "  
 + zlt.withZoneSameInstant(ZoneId.of(zoneId))  
 .format(formatter));  
 });  
  
 return result;  
}Copy

An output snapshot of this method can be as follows:

2019-Feb-26T14:26:30 PM +0200 in Africa/Nairobi   
 is 2019-Feb-26T15:26:30 PM +0300  
2019-Feb-26T14:26:30 PM +0200 in America/Marigot   
 is 2019-Feb-26T08:26:30 AM -0400  
...  
2019-Feb-26T14:26:30 PM +0200 in Pacific/Samoa   
 is 2019-Feb-26T01:26:30 AM -1100Copy

68. Displaying date-time information about a flight

The solution that is presented in this section will display the following information about the 15 hours and 30 minutes flight from Perth, Australia to Bucharest, Europe:

* UTC date-time at departure and arrival
* Perth date-time at departure and arrival in Bucharest
* Bucharest date-time at departure and arrival

Let's assume that the reference departure date-time from Perth is February 26, 2019, at 16:00 (or 4:00 PM):

LocalDateTime ldt = LocalDateTime.of(  
 2019, Month.FEBRUARY, 26, 16, 00);Copy

First, let's combines this date-time with the time zone of Australia/Perth (+08:00). This will result in a ZonedDateTime object that is specific to Australia/Perth (this is the clock date and time in Perth at departure):

// 04:00 PM, Feb 26, 2019 +0800 Australia/Perth  
ZonedDateTime auPerthDepart   
 = ldt.atZone(ZoneId.of("Australia/Perth"));Copy

Further, let's add 15 hours and 30 minutes to ZonedDateTime. The resulting ZonedDateTime represents the date-time in Perth (this is the clock date and time in Perth on arrival in Bucharest):

// 07:30 AM, Feb 27, 2019 +0800 Australia/Perth  
ZonedDateTime auPerthArrive   
 = auPerthDepart.plusHours(15).plusMinutes(30);Copy

Now, let's calculate the date-time in Bucharest at the departure date-time in Perth. Basically, the following code expresses the departure date-time from the Perth time zone in the Bucharest time zone:

// 10:00 AM, Feb 26, 2019 +0200 Europe/Bucharest  
ZonedDateTime euBucharestDepart   
 = auPerthDepart.withZoneSameInstant(ZoneId.of("Europe/Bucharest"));Copy

Finally, let's calculate the date-time in Bucharest on arrival. The following code expresses the arrival date-time from the Perth time zone in the Bucharest time zone:

// 01:30 AM, Feb 27, 2019 +0200 Europe/Bucharest  
ZonedDateTime euBucharestArrive   
 = auPerthArrive.withZoneSameInstant(ZoneId.of("Europe/Bucharest"));Copy

As shown in the following figure, the UTC time at departure from Perth is 8:00 AM, while the UTC time on arrival in Bucharest is 11:30 PM:

Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente

These times can be easily extracted as OffsetDateTime, as follows:

// 08:00 AM, Feb 26, 2019  
OffsetDateTime utcAtDepart = auPerthDepart.withZoneSameInstant(  
 ZoneId.of("UTC")).toOffsetDateTime();  
  
// 11:30 PM, Feb 26, 2019  
OffsetDateTime utcAtArrive = auPerthArrive.withZoneSameInstant(  
 ZoneId.of("UTC")).toOffsetDateTime();Copy

69. Converting a Unix timestamp to date-time

For this solution, let's suppose the following Unix timestamp—1573768800. This timestamp is equivalent to the following:

* 11/14/2019 @ 10:00pm (UTC)
* 2019-11-14T22:00:00+00:00 in ISO-8601
* Thu, 14 Nov 2019 22:00:00 +0000 in RFC 822, 1036, 1123, 2822
* Thursday, 14-Nov-19 22:00:00 UTC in RFC 2822
* 2019-11-14T22:00:00+00:00 in RFC 3339

In order to convert a Unix timestamp to a date-time, it is important to know that the Unix timestamps resolution is in seconds, while java.util.Date needs milliseconds. So, the solution to obtain a Date object from a Unix timestamp requires a conversion from seconds to milliseconds by multiplying the Unix timestamp by 1,000 as shown in the following two examples:

long unixTimestamp = 1573768800;  
  
// Fri Nov 15 00:00:00 EET 2019 - in the default time zone  
Date date = new Date(unixTimestamp \* 1000L);  
  
// Fri Nov 15 00:00:00 EET 2019 - in the default time zone  
Date date = new Date(TimeUnit.MILLISECONDS  
 .convert(unixTimestamp, TimeUnit.SECONDS));Copy

Starting with JDK 8, the Date class uses the from(Instant instant) method. Moreover, the Instant class comes with the ofEpochSecond(long epochSecond) method, which returns an instance of Instant, using the given seconds from the epoch, of 1970-01-01T00:00:00Z:

// 2019-11-14T22:00:00Z in UTC  
Instant instant = Instant.ofEpochSecond(unixTimestamp);  
  
// Fri Nov 15 00:00:00 EET 2019 - in the default time zone  
Date date = Date.from(instant);Copy

The instant that was obtained in the previous example can be used to create LocalDateTime or ZonedDateTime, as follows:

// 2019-11-15T06:00  
LocalDateTime date = LocalDateTime  
 .ofInstant(instant, ZoneId.of("Australia/Perth"));  
  
// 2019-Nov-15 00:00:00 +0200 Europe/Bucharest  
ZonedDateTime date = ZonedDateTime  
 .ofInstant(instant, ZoneId.of("Europe/Bucharest"));Copy

70. Finding the first/last day of the month

The proper solution to this problem will rely on JDK 8's, Temporal and TemporalAdjuster interfaces.

The Temporal interface sits behind representations of date-time. In other words, classes that represent a date and/or a time implement this interface. For example, the following classes are just a few that implement this interface:

* LocalDate (date without a time zone in the ISO-8601 calendar system)
* LocalTime (time without a time zone in the ISO-8601 calendar system)
* LocalDateTime (date-time without a time zone in the ISO-8601 calendar system)
* ZonedDateTime (date-time with a time zone in the ISO-8601 calendar system), and so on
* OffsetDateTime (date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system)
* HijrahDate (date in the Hijrah calendar system)

The TemporalAdjuster class is a functional interface that defines strategies that can be used to adjust a Temporal object. Beside the possibility of defining custom strategies, the TemporalAdjuster class provides several predefined strategies, as follows (the documentation contains the entire list, which is pretty impressive):

* firstDayOfMonth() (return the first day of the current month)
* lastDayOfMonth() (return the last day of the current month)
* firstDayOfNextMonth() (return the first day of the next month)
* firstDayOfNextYear() (return the first day of the next year)

Notice that the first two adjusters in the preceding list are exactly the ones needed by this problem.

Consider a fix—LocalDate:

LocalDate date = LocalDate.of(2019, Month.FEBRUARY, 27);Copy

And, let's see when the first/last days of February are:

// 2019-02-01  
LocalDate firstDayOfFeb   
 = date.with(TemporalAdjusters.firstDayOfMonth());  
  
// 2019-02-28  
LocalDate lastDayOfFeb   
 = date.with(TemporalAdjusters.lastDayOfMonth());Copy

Looks like relying on the predefined strategies is pretty simple. But, let's assume that the problem requests you to find the date that's 21 days after February, 27 2019, which is March 20, 2019. For this problem there is no predefined strategy, therefore a custom strategy is needed. A solution to this problem can rely on a lambda expression, as in the following helper method:

public static LocalDate getDayAfterDays(  
 LocalDate startDate, int days) {  
  
 Period period = Period.ofDays(days);  
 TemporalAdjuster ta = p -> p.plus(period);  
 LocalDate endDate = startDate.with(ta);  
  
 return endDate;  
}Copy

If this method lives in a class named DateTimes, then the following call will return the expected result:

// 2019-03-20  
LocalDate datePlus21Days = DateTimes.getDayAfterDays(date, 21);Copy

Following the same technique, but relying on the static factory method ofDateAdjuster(), the following snippet of code defines a static adjuster that returns the next date that falls on a Saturday:

static TemporalAdjuster NEXT\_SATURDAY   
 = TemporalAdjusters.ofDateAdjuster(today -> {  
  
 DayOfWeek dayOfWeek = today.getDayOfWeek();  
  
 if (dayOfWeek == DayOfWeek.SATURDAY) {  
 return today;  
 }  
  
 if (dayOfWeek == DayOfWeek.SUNDAY) {  
 return today.plusDays(6);  
 }  
  
 return today.plusDays(6 - dayOfWeek.getValue());  
});Copy

Let's call this method for February 27, 2019 (the next Saturday is on March 2, 2019):

// 2019-03-02  
LocalDate nextSaturday = date.with(NEXT\_SATURDAY);Copy

Finally, this functional interface defines an abstract method named adjustInto(). This method can be overridden in custom implementations by passing a Temporal object to it, as follows:

public class NextSaturdayAdjuster implements TemporalAdjuster {  
  
 @Override  
 public Temporal adjustInto(Temporal temporal) {  
  
 DayOfWeek dayOfWeek = DayOfWeek  
 .of(temporal.get(ChronoField.DAY\_OF\_WEEK));  
  
 if (dayOfWeek == DayOfWeek.SATURDAY) {  
 return temporal;  
 }  
  
 if (dayOfWeek == DayOfWeek.SUNDAY) {  
 return temporal.plus(6, ChronoUnit.DAYS);  
 }  
  
 return temporal.plus(6 - dayOfWeek.getValue(), ChronoUnit.DAYS);  
 }  
}Copy

Here is the usage example:

NextSaturdayAdjuster nsa = new NextSaturdayAdjuster();  
  
// 2019-03-02  
LocalDate nextSaturday = date.with(nsa);Copy

71. Defining/extracting zone offsets

By *zone offset*, we understand the amount of time needed to be added/subtracted from the GMT/UTC time in order to obtain the date-time for a specific zone on the globe (for example, Perth, Australia). Commonly, a zone offset is printed as a fixed number of hour and minutes: +02:00, -08:30, +0400, UTC+01:00, and so on.

So, in short, a zone offset is the amount of time by which a time zone differs from GMT/UTC.

Before JDK 8

Before JDK 8, a time zone can be defined via java.util.TimeZone. With this time zone, the code can obtain the zone offset via the TimeZone.getRawOffset() method (the *raw* part comes from the fact that this method doesn't take into account Daylight Saving Time). The code that is bundled to this book contains this solution.

Starting with JDK 8

Starting with JDK 8, there are two classes responsible for dealing with time zone representations. First, there is java.time.ZoneId, which represents a time zone such as Athens, Europe, and second there is java.time.ZoneOffset (extends ZoneId), which represents the fixed amount of time (offset) of the specified time zone with GMT/UTC.

The new Java date-time API deals with Daylight Saving Time by default; therefore, a region with summer-winter cycles that uses Daylight Saving Time will have two ZoneOffset classes.

The UTC zone offset can be easily obtained as follows (this is +00:00, represented in Java by the Z character):

// Z  
ZoneOffset zoneOffsetUTC = ZoneOffset.UTC;Copy

The system default time zone can also be obtained via the ZoneOffset class:

// Europe/Athens  
ZoneId defaultZoneId = ZoneOffset.systemDefault();Copy

In order to take the zone offset with Daylight Saving Time, the code needs to associate a date-time with it. For example, associate a LocalDateTime class (Instant can also be used) like this:

// by default it deals with the Daylight Saving Times  
LocalDateTime ldt = LocalDateTime.of(2019, 6, 15, 0, 0);  
ZoneId zoneId = ZoneId.of("Europe/Bucharest");  
  
// +03:00  
ZoneOffset zoneOffset = zoneId.getRules().getOffset(ldt);Copy

A zone offset can also be obtained from a string. For example, the following code obtains a zone offset of +02:00:

ZoneOffset zoneOffsetFromString = ZoneOffset.of("+02:00");Copy

This is a very convenient approach of quickly adding a zone offset to a Temporal object that supports zone offsets. For example, use it to add a zone offset to OffsetTime and OffsetDateTime (convenient ways for storing a date in a database, or sending over the wires):

OffsetTime offsetTime = OffsetTime.now(zoneOffsetFromString);  
OffsetDateTime offsetDateTime   
 = OffsetDateTime.now(zoneOffsetFromString);Copy

Another solution to our problem is to rely on defining ZoneOffset from hours, minutes, and seconds. One of the helper methods of ZoneOffset is dedicated to this:

// +08:30 (this was obtained from 8 hours and 30 minutes)  
ZoneOffset zoneOffsetFromHoursMinutes   
 = ZoneOffset.ofHoursMinutes(8, 30);Copy

Next to ZoneOffset.ofHoursMinutes(), there is ZoneOffset.ofHours(), ofHoursMinutesSeconds() and ofTotalSeconds().

Finally, every Temporal object that supports a zone offset provides a handy getOffset() method. For example, the following code gets the zone offset from the preceding offsetDateTime object:

// +02:00  
ZoneOffset zoneOffsetFromOdt = offsetDateTime.getOffset();Copy

72. Converting between Date and Temporal

The solution that is presented here will cover the following Temporal classes—Instant, LocalDate, LocalDateTime, ZonedDateTime, OffsetDateTime, LocalTime, and OffsetTime.

Date – Instant

In order to convert from Date to Instant, the solution can rely on the Date.toInstant() method. The reverse can be accomplished via the Date.from(Instant instant) method:

* Date to Instant can be accomplished like this:

Date date = new Date();  
  
// e.g., 2019-02-27T12:02:49.369Z, UTC  
Instant instantFromDate = date.toInstant();Copy

* Instant to Date can be accomplished like this:

Instant instant = Instant.now();  
  
// Wed Feb 27 14:02:49 EET 2019, default system time zone  
Date dateFromInstant = Date.from(instant);Copy

Keep in mind that Date is not time-zone aware, but it is displayed in the system default time zone (for example, via toString()). Instant is with a UTC time zone.

Let's quickly wrap these snippets of code in two utility methods, defined in a utility class—DateConverters:

public static Instant dateToInstant(Date date) {  
  
 return date.toInstant();  
}  
  
public static Date instantToDate(Instant instant) {  
  
 return Date.from(instant);  
}Copy

Further, let's enrich this class with the methods from the following screenshot:

Texto

Descripción generada automáticamente

The constant from the screenshot, DEFAULT\_TIME\_ZONE, is the system default time zone:

public static final ZoneId DEFAULT\_TIME\_ZONE = ZoneId.systemDefault();Copy

Date – LocalDate

A Date object can be converted to LocalDate via an Instant object. Once we have obtained the Instant object from the given Date object, the solution can apply to it the system default time zone, and call the toLocaleDate() method:

// e.g., 2019-03-01  
public static LocalDate dateToLocalDate(Date date) {  
  
 return dateToInstant(date).atZone(DEFAULT\_TIME\_ZONE).toLocalDate();  
}Copy

Converting from LocalDate to Date should take into account that LocalDate doesn't contain a time component as Date, so the solution must supply a time component as the start of the day (more details regarding this can be found in the S*tart and end of a day* problem):

// e.g., Fri Mar 01 00:00:00 EET 2019  
public static Date localDateToDate(LocalDate localDate) {  
  
 return Date.from(localDate.atStartOfDay(  
 DEFAULT\_TIME\_ZONE).toInstant());  
}Copy

Date – DateLocalTime

Converting from Date to DateLocalTime is the same as converting from Date to LocalDate, apart from the fact that the solution should call the toLocalDateTime() method as follows:

// e.g., 2019-03-01T07:25:25.624  
public static LocalDateTime dateToLocalDateTime(Date date) {  
  
 return dateToInstant(date).atZone(  
 DEFAULT\_TIME\_ZONE).toLocalDateTime();  
}Copy

Converting from LocalDateTime to Date is straightforward. Just apply the system default time zone and call toInstant():

// e.g., Fri Mar 01 07:25:25 EET 2019  
public static Date localDateTimeToDate(LocalDateTime localDateTime) {  
  
 return Date.from(localDateTime.atZone(  
 DEFAULT\_TIME\_ZONE).toInstant());  
}Copy

Date – ZonedDateTime

Converting Date to ZonedDateTime can be accomplished via the Instant object obtained from the given Date object and the system default time zone:

// e.g., 2019-03-01T07:25:25.624+02:00[Europe/Athens]  
public static ZonedDateTime dateToZonedDateTime(Date date) {  
  
 return dateToInstant(date).atZone(DEFAULT\_TIME\_ZONE);  
}Copy

Converting ZonedDateTime to Date is just about converting ZonedDateTime to Instant:

// e.g., Fri Mar 01 07:25:25 EET 2019  
public static Date zonedDateTimeToDate(ZonedDateTime zonedDateTime) {  
  
 return Date.from(zonedDateTime.toInstant());  
}Copy

Date – OffsetDateTime

Converting from Date to OffsetDateTime relies on the toOffsetDateTime() method:

// e.g., 2019-03-01T07:25:25.624+02:00  
public static OffsetDateTime dateToOffsetDateTime(Date date) {  
  
 return dateToInstant(date).atZone(  
 DEFAULT\_TIME\_ZONE).toOffsetDateTime();  
}Copy

An approach for converting from OffsetDateTime to Date requires two steps. First, convert OffsetDateTime to LocalDateTime. Second, convert LocalDateTime to Instant with the corresponding offset:

// e.g., Fri Mar 01 07:55:49 EET 2019  
public static Date offsetDateTimeToDate(  
 OffsetDateTime offsetDateTime) {  
  
 return Date.from(offsetDateTime.toLocalDateTime()  
 .toInstant(ZoneOffset.of(offsetDateTime.getOffset().getId())));  
}Copy

Date – LocalTime

Converting Date to LocalTime can rely on the LocalTime.toInstant() method as follows:

// e.g., 08:03:20.336  
public static LocalTime dateToLocalTime(Date date) {  
  
 return LocalTime.ofInstant(dateToInstant(date), DEFAULT\_TIME\_ZONE);  
}Copy

Converting LocalTime to Date should take into account that LocalTime doesn't have a date component. This means that the solution should set the date on January 1, 1970, the epoch:

// e.g., Thu Jan 01 08:03:20 EET 1970  
public static Date localTimeToDate(LocalTime localTime) {  
  
 return Date.from(localTime.atDate(LocalDate.EPOCH)  
 .toInstant(DEFAULT\_TIME\_ZONE.getRules()  
 .getOffset(Instant.now())));  
}Copy

Date – OffsetTime

Converting Date to OffsetTime can rely on the OffsetTime.toInstant() method as follows:

// e.g., 08:03:20.336+02:00  
public static OffsetTime dateToOffsetTime(Date date) {  
  
 return OffsetTime.ofInstant(dateToInstant(date), DEFAULT\_TIME\_ZONE);  
}Copy

Converting OffsetTime to Date should take into account that OffsetTime doesn't have a date component. This means that the solution should set the date at January 1, 1970, the epoch:

// e.g., Thu Jan 01 08:03:20 EET 1970  
public static Date offsetTimeToDate(OffsetTime offsetTime) {  
  
 return Date.from(offsetTime.atDate(LocalDate.EPOCH).toInstant());  
}Copy

73. Iterating a range of dates

Let's assume that the range is demarcated by the start date, 2019 Feb 1, and the end date, 2019 Feb 21. The solution to this problem should loop the [2019 Feb 1, 2019 Feb 21) interval with a step of a day and print each date on the screen. Basically, there are two main problems to solve:

* Stop looping once the start date is equal with the end date.
* Increase the start date day by day until the end date.

Before JDK 8

Before JDK 8, the solution can rely on the Calendar utility class. The code that is bundled to this book contains this solution.

Starting with JDK 8

First, starting with JDK 8, the dates can be easily defined as LocalDate, without the help of Calendar:

LocalDate startLocalDate = LocalDate.of(2019, 2, 1);  
LocalDate endLocalDate = LocalDate.of(2019, 2, 21);Copy

Once the start date is equal with the end date, we stop the loop via the LocalDate.isBefore(ChronoLocalDate other) method. This flag method checks if this date is before the given date.

Increasing the start date day by day until the end date can be accomplished using the LocalDate.plusDays(long daysToAdd) method. Using these two methods in a for loop results in the following code:

for (LocalDate date = startLocalDate;   
 date.isBefore(endLocalDate); date = date.plusDays(1)) {  
  
 // do something with this day  
 System.out.println(date);  
}Copy

A snapshot of the output should be as follows:

2019-02-01  
2019-02-02  
2019-02-03  
...  
2019-02-20Copy

Starting with JDK 9

JDK 9 can solve this problem using a single line of code. This is possible thanks to the new LocalDate.datesUntil(LocalDate endExclusive) method. This method returns Stream<LocalDate> with an incremental step of one day:

startLocalDate.datesUntil(endLocalDate).forEach(System.out::println);Copy

If the incremental step should be expressed in days, weeks, months, or years, then rely on LocalDate.datesUntil(LocalDate endExclusive, Period step). For example, an incremental step of 1 week can be specified as follows:

startLocalDate.datesUntil(endLocalDate, Period.ofWeeks(1)).forEach(System.out::println);Copy

The output should be (weeks 1-8, weeks 8-15) as follows:

2019-02-01  
2019-02-08  
2019-02-15Copy

74. Calculating age

Probably the most commonly used case of difference between two dates is about calculating the age of a person. Typically, the age of a person is expressed in years, but sometimes months, and even days, should be provided.

Before JDK 8

Before JDK 8, trying to provide a good solution can rely on Calendar and/or SimpleDateFormat. The code that is bundled to this book contains such a solution.

Starting with JDK 8

A better idea is to upgrade to JDK 8, and rely on the following straightforward snippet of code:

LocalDate startLocalDate = LocalDate.of(1977, 11, 2);  
LocalDate endLocalDate = LocalDate.now();  
  
long years = ChronoUnit.YEARS.between(startLocalDate, endLocalDate);Copy

Adding months and days to the result is also easy to accomplish, thanks to the Period class:

Period periodBetween = Period.between(startLocalDate, endLocalDate);Copy

Now, the age in years, months, and days can be obtained via periodBetween.getYears(), periodBetween.getMonths(), and periodBetween.getDays().

For example, between the current date, February 28, 2019, and November 2, 1977, we have 41 years, 3 months, and 26 days.

75. Start and end of a day

In JDK 8, trying to find the start/end of a day can be accomplished in several ways.

Let's consider a day expressed via LocalDate:

LocalDate localDate = LocalDate.of(2019, 2, 28);Copy

The solution to finding the start of the day February 28, 2019, relies on a method named atStartOfDay(). This method returns LocalDateTime from this date at the time of midnight, 00:00:

// 2019-02-28T00:00  
LocalDateTime ldDayStart = localDate.atStartOfDay();Copy

Alternatively, the solution can use the of(LocalDate date, LocalTime time) method. This method combines the given date and time into LocalDateTime. So, if the passed time is LocalTime.MIN (the time of midnight at the start of the day) then the result will be as follows:

// 2019-02-28T00:00  
LocalDateTime ldDayStart = LocalDateTime.of(localDate, LocalTime.MIN);Copy

The end of the day of a LocalDate object can be obtained using at least two solutions. One solution consist of relying on LocalDate.atTime(LocalTime time). The resulting LocalDateTime can represent the combination of this date with the end of the day, if the solution passes as an argument, LocalTime.MAX (the time just before midnight at the end of the day):

// 2019-02-28T23:59:59.999999999  
LocalDateTime ldDayEnd = localDate.atTime(LocalTime.MAX);Copy

Alternatively, the solution can combine LocalTime.MAX with the given date, via the atDate(LocalDate date) method:

// 2019-02-28T23:59:59.999999999  
LocalDateTime ldDayEnd = LocalTime.MAX.atDate(localDate);Copy

Since LocalDate doesn't have the concept of a time zone, the preceding examples are prone to issues caused by different corner-cases, for example, Daylight Saving Time. Some Daylight Saving Times impose a change of hour at midnight (00:00 becomes 01:00 AM), which means that the start of the day is at 01:00:00, not at 00:00:00. In order to mitigate these issues, consider the following examples that extend the preceding examples to use ZonedDateTime, which is Daylight Saving Time aware:

// 2019-02-28T00:00+08:00[Australia/Perth]  
ZonedDateTime ldDayStartZone   
 = localDate.atStartOfDay(ZoneId.of("Australia/Perth"));  
  
// 2019-02-28T00:00+08:00[Australia/Perth]  
ZonedDateTime ldDayStartZone = LocalDateTime  
 .of(localDate, LocalTime.MIN).atZone(ZoneId.of("Australia/Perth"));  
  
// 2019-02-28T23:59:59.999999999+08:00[Australia/Perth]  
ZonedDateTime ldDayEndZone = localDate.atTime(LocalTime.MAX)  
 .atZone(ZoneId.of("Australia/Perth"));  
  
// 2019-02-28T23:59:59.999999999+08:00[Australia/Perth]  
ZonedDateTime ldDayEndZone = LocalTime.MAX.atDate(localDate)  
 .atZone(ZoneId.of("Australia/Perth"));Copy

Now, let's consider the following—LocalDateTime, February 28, 2019, 18:00:00:

LocalDateTime localDateTime = LocalDateTime.of(2019, 2, 28, 18, 0, 0);Copy

The obvious solution is to extract LocalDate from LocalDateTime and apply the previous approaches. Another solution relies on the fact that every implementation of the Temporal interface (including LocalDate) can take advantage of the with(TemporalField field, long newValue) method. Mainly, the with() method returns a copy of this date with the specified field, ChronoField, set to newValue. So, if the solution sets ChronoField.NANO\_OF\_DAY (nanoseconds of a day) as LocalTime.MIN, then the result will be the start of the day. The trick here is to convert LocalTime.MIN to nanoseconds via toNanoOfDay(), as follows:

// 2019-02-28T00:00  
LocalDateTime ldtDayStart = localDateTime  
 .with(ChronoField.NANO\_OF\_DAY, LocalTime.MIN.toNanoOfDay());Copy

This is equivalent to the following:

LocalDateTime ldtDayStart   
 = localDateTime.with(ChronoField.HOUR\_OF\_DAY, 0);Copy

The end of the day is pretty similar. Just pass LocalTime.MAX instead of MIN:

// 2019-02-28T23:59:59.999999999  
LocalDateTime ldtDayEnd = localDateTime  
 .with(ChronoField.NANO\_OF\_DAY, LocalTime.MAX.toNanoOfDay());Copy

This is equivalent to the following:

LocalDateTime ldtDayEnd = localDateTime.with(  
 ChronoField.NANO\_OF\_DAY, 86399999999999L);Copy

Like LocalDate, the LocalDateTime object is not aware of time zones. In this case, ZonedDateTime can help:

// 2019-02-28T00:00+08:00[Australia/Perth]  
ZonedDateTime ldtDayStartZone = localDateTime  
 .with(ChronoField.NANO\_OF\_DAY, LocalTime.MIN.toNanoOfDay())  
 .atZone(ZoneId.of("Australia/Perth"));  
  
// 2019-02-28T23:59:59.999999999+08:00[Australia/Perth]  
ZonedDateTime ldtDayEndZone = localDateTime  
 .with(ChronoField.NANO\_OF\_DAY, LocalTime.MAX.toNanoOfDay())  
 .atZone(ZoneId.of("Australia/Perth"));Copy

As a bonus here, let's see the start/end of the day with UTC. Beside the solution relying on the with() method, another solution can rely on toLocalDate(), as follows:

// e.g., 2019-02-28T09:23:10.603572Z  
ZonedDateTime zdt = ZonedDateTime.now(ZoneOffset.UTC);  
  
// 2019-02-28T00:00Z  
ZonedDateTime dayStartZdt   
 = zdt.toLocalDate().atStartOfDay(zdt.getZone());  
  
// 2019-02-28T23:59:59.999999999Z  
ZonedDateTime dayEndZdt = zdt.toLocalDate()  
 .atTime(LocalTime.MAX).atZone(zdt.getZone());Copy

Because of the numerous issues with java.util.Date and Calendar, it is advisable to avoid trying to implement a solution to this problem with them.

76. Difference between two dates

Computing the difference between two dates is a very common task (for example, see the *Calculating age* section). Let's see a collection of other approaches that can be used to obtain the difference between two dates in milliseconds, seconds, hours, and so on.

Before JDK 8

The recommended way to represent date-time information is via the java.util.Date and Calendar classes. The easiest difference to compute is expressed in milliseconds. The code that is bundled to this book contains such a solution.

Starting with JDK 8

Starting with JDK 8, the recommended way to represent date-time information is via Temporal (for example, DateTime, DateLocalTime, ZonedDateTime, and so on).

Let's assume the following two LocalDate objects, January 1, 2018, and March 1, 2019:

LocalDate ld1 = LocalDate.of(2018, 1, 1);  
LocalDate ld2 = LocalDate.of(2019, 3, 1);Copy

The simplest way to compute the difference between these two Temporal objects is via the ChronoUnit class. Beside representing the standard set of date periods units, ChronoUnit comes with several handy methods, including between(Temporal t1Inclusive, Temporal t2Exclusive). As its name suggests, the between() method calculates the amount of time between two Temporal objects. Let's see it at work to compute the difference between ld1 and ld2 in days, months, and years:

// 424  
long betweenInDays = Math.abs(ChronoUnit.DAYS.between(ld1, ld2));  
  
// 14  
long betweenInMonths = Math.abs(ChronoUnit.MONTHS.between(ld1, ld2));  
  
// 1  
long betweenInYears = Math.abs(ChronoUnit.YEARS.between(ld1, ld2));Copy

Alternatively, every Temporal exposes a method named until(). Actually, LocalDate has two, one that returns Period as a difference between two dates and another one that returns long as a difference between two dates in the specified time unit. Using the one that returns Period looks like this:

Period period = ld1.until(ld2);  
  
// Difference as Period: 1y2m0d  
System.out.println("Difference as Period: " + period.getYears() + "y"   
 + period.getMonths() + "m" + period.getDays() + "d");Copy

Using the one that allows us to specify the time unit looks like this:

// 424  
long untilInDays = Math.abs(ld1.until(ld2, ChronoUnit.DAYS));  
  
// 14  
long untilInMonths = Math.abs(ld1.until(ld2, ChronoUnit.MONTHS));  
  
// 1  
long untilInYears = Math.abs(ld1.until(ld2, ChronoUnit.YEARS));Copy

The ChronoUnit.convert() method is also useful in the case of LocalDateTime. Let's consider the following two LocalDateTime objects—January 1, 2018 22:15:15, and March 1, 2019 23:15:15:

LocalDateTime ldt1 = LocalDateTime.of(2018, 1, 1, 22, 15, 15);  
LocalDateTime ldt2 = LocalDateTime.of(2018, 1, 1, 23, 15, 15);Copy

Now, let's see the difference between ldt1 and ldt2, when expressed in minutes:

// 60  
long betweenInMinutesWithoutZone   
 = Math.abs(ChronoUnit.MINUTES.between(ldt1, ldt2));Copy

And, the difference when expressed in hours via the LocalDateTime.until() method:

// 1  
long untilInMinutesWithoutZone   
 = Math.abs(ldt1.until(ldt2, ChronoUnit.HOURS));Copy

But, a really awesome thing about ChronoUnit.between() and until() is the fact that they work with ZonedDateTime. For example, let's consider ldt1 in the Europe/Bucharest time zone and in the Australia/Perth time zone, plus one hour:

ZonedDateTime zdt1 = ldt1.atZone(ZoneId.of("Europe/Bucharest"));  
ZonedDateTime zdt2 = zdt1.withZoneSameInstant(  
 ZoneId.of("Australia/Perth")).plusHours(1);Copy

Now, let's use ChronoUnit.between() to express the difference between zdt1 and zdt2 in minutes, and ZonedDateTime.until() to express the difference between zdt1 and zdt2 in hours:

// 60  
long betweenInMinutesWithZone   
 = Math.abs(ChronoUnit.MINUTES.between(zdt1, zdt2));  
  
// 1  
long untilInHoursWithZone   
 = Math.abs(zdt1.until(zdt2, ChronoUnit.HOURS));Copy

Finally, let's repeat this technique, but for two independent ZonedDateTime objects; one obtained for ldt1 and one for ldt2:

ZonedDateTime zdt1 = ldt1.atZone(ZoneId.of("Europe/Bucharest"));  
ZonedDateTime zdt2 = ldt2.atZone(ZoneId.of("Australia/Perth"));  
  
// 300  
long betweenInMinutesWithZone   
 = Math.abs(ChronoUnit.MINUTES.between(zdt1, zdt2));  
  
// 5  
long untilInHoursWithZone   
 = Math.abs(zdt1.until(zdt2, ChronoUnit.HOURS));Copy

77. Implementing a chess clock

Starting with JDK 8, the java.time package has an abstract class named Clock. The main purpose of this class is to allow us to plug in different clocks when needed (for example, for testing purposes). By default, Java comes with four implementations: SystemClock, OffsetClock, TickClock, and FixedClock. For each of these implementations, there are static methods in the Clock class. For example, the following code creates FixedClock (a clock that always returns the same Instant):

Clock fixedClock = Clock.fixed(Instant.now(), ZoneOffset.UTC);Copy

There is also TickClock, which returns the current Instant ticking in whole seconds for the given time zone:

Clock tickClock = Clock.tickSeconds(ZoneId.of("Europe/Bucharest"));Copy

There is also a method that can be used to tick in whole minutes, tickMinutes(), and a generic one, tick(), which allows us to specify Duration.

A Clock class may also support time zones and offsets, but the most important method of a Clock class is instant(). This method returns the instant of Clock:

// 2019-03-01T13:29:34Z  
System.out.println(tickClock.instant());Copy

There is also the millis() method, which returns the current instant of the clock in milliseconds.

Let's assume that we want to implement a clock that acts a chess clock:

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza baja

In order to implement a Clock class, there are several steps to follow:

1. Extend the Clock class.
2. Implement Serializable.
3. Override at least the abstract methods inherited from Clock.

A skeleton of a Clock class is as follows:

public class ChessClock extends Clock implements Serializable {  
  
 @Override  
 public ZoneId getZone() {  
 ...  
 }  
  
 @Override  
 public Clock withZone(ZoneId zone) {  
 ...  
 }  
  
 @Override  
 public Instant instant() {  
 ...  
 }  
}Copy

Our ChessClock will work only with UTC; no other time zone will be supported. This means that the getZone() and withZone() methods can be implemented as follows (of course, this can be modified in the future):

@Override  
public ZoneId getZone() {  
 return ZoneOffset.UTC;  
}  
  
@Override  
public Clock withZone(ZoneId zone) {  
 throw new UnsupportedOperationException(  
 "The ChessClock works only in UTC time zone");  
}Copy

The climax of our implementation is the instant() method. The difficulty consists in managing two Instant, one for the player from the left (instantLeft) and one for the player from the right (instantRight). We can associate every call of the instant() method with the fact that the current player has performed a move, and now it is the other player's turn. So, basically, this logic says that the same player cannot call instant() twice. Implementing this logic, the instant() method is as follows:

public class ChessClock extends Clock implements Serializable {  
  
 public enum Player {  
 LEFT,  
 RIGHT  
 }  
  
 private static final long serialVersionUID = 1L;  
  
 private Instant instantStart;  
 private Instant instantLeft;  
 private Instant instantRight;  
 private long timeLeft;  
 private long timeRight;  
 private Player player;  
  
 public ChessClock(Player player) {  
 this.player = player;  
 }  
  
 public Instant gameStart() {  
  
 if (this.instantStart == null) {  
 this.timeLeft = 0;  
 this.timeRight = 0;  
 this.instantStart = Instant.now();  
 this.instantLeft = instantStart;  
 this.instantRight = instantStart;  
 return instantStart;  
 }  
  
 throw new IllegalStateException(  
 "Game already started. Stop it and try again.");  
 }  
  
 public Instant gameEnd() {  
  
 if (this.instantStart != null) {  
 instantStart = null;  
 return Instant.now();  
 }  
  
 throw new IllegalStateException("Game was not started.");  
 }  
  
 @Override  
 public ZoneId getZone() {  
 return ZoneOffset.UTC;  
 }  
  
 @Override  
 public Clock withZone(ZoneId zone) {  
 throw new UnsupportedOperationException(  
 "The ChessClock works only in UTC time zone");  
 }  
  
 @Override  
 public Instant instant() {  
  
 if (this.instantStart != null) {  
 if (player == Player.LEFT) {  
 player = Player.RIGHT;  
  
 long secondsLeft = Instant.now().getEpochSecond()   
 - instantRight.getEpochSecond();  
 instantLeft = instantLeft.plusSeconds(  
 secondsLeft - timeLeft);  
 timeLeft = secondsLeft;  
  
 return instantLeft;  
 } else {  
 player = Player.LEFT;  
  
 long secondsRight = Instant.now().getEpochSecond()   
 - instantLeft.getEpochSecond();  
 instantRight = instantRight.plusSeconds(  
 secondsRight - timeRight);  
 timeRight = secondsRight;  
  
 return instantRight;  
 }  
 }  
  
 throw new IllegalStateException("Game was not started.");  
 }  
}Copy

So, depending on which player calls the instant() method, the code computes the number of seconds needed by that player to think until she/he performed a move. Moreover, the code switches the player, so the next call of instant() will deal with the other player.

Let's consider a chess game starting at 2019-03-01T14:02:46.309459Z:

ChessClock chessClock = new ChessClock(Player.LEFT);  
  
// 2019-03-01T14:02:46.309459Z  
Instant start = chessClock.gameStart();Copy

Further, the players perform the following sequence of movements until the player from the right wins the game:

Left moved first after 2 seconds: 2019-03-01T14:02:48.309459Z  
Right moved after 5 seconds: 2019-03-01T14:02:51.309459Z  
Left moved after 6 seconds: 2019-03-01T14:02:54.309459Z  
Right moved after 1 second: 2019-03-01T14:02:52.309459Z  
Left moved after 2 second: 2019-03-01T14:02:56.309459Z  
Right moved after 3 seconds: 2019-03-01T14:02:55.309459Z  
Left moved after 10 seconds: 2019-03-01T14:03:06.309459Z  
Right moved after 11 seconds and win: 2019-03-01T14:03:06.309459ZCopy

It looks like the clock has correctly registered the movements of the players.

Finally, the game is over after 40 seconds:

Game ended:2019-03-01T14:03:26.350749300Z  
Instant end = chessClock.gameEnd();  
  
Game duration: 40 seconds  
// Duration.between(start, end).getSeconds();Copy

Summary

Mission accomplished! This chapter provided a comprehensive overview of working with date and time information. A wide range of applications must manipulate this kind of information. Therefore, having the solutions to these problems under your tool belt is not optional. From Date and Calendar to LocalDate, LocalTime, LocalDateTime, ZoneDateTime, OffsetDateTime, OffsetTime, and Instant—they are all important and very useful in daily tasks that involve date and time.

Download the applications from this chapter to see the results and to see additional details.

Type Inference

This chapter includes 21 problems that involve JEP 286 or Java **Local Variable Type Inference** (**LVTI**), also known as the var type. These problems have been carefully crafted to reveal the best practices and common mistakes that are involved in using var. By the end of this chapter, you will have learned everything you need to know about var to push it to production.

Problems

Use the following problems to test your type inference programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Simple** var **example**: Write a program that exemplifies the correct usage of type inference (var) with respect to the code's readability.
2. **Using var with primitive types**: Write a program that exemplifies the usage of var with Java primitive types (int, long, float, and double).
3. **Using var and implicit type casting to sustain the code's maintainability**: Write a program that exemplifies how var and *implicit type casting* can sustain the code's maintainability.
4. **Explicit downcast or better avoid** var: Write a program that exemplifies the combination of var and explicit downcast and explain why var should be avoided.
5. **Avoid using var if the called names don't contain enough type information for humans**: Provide examples where var should be avoided because its combination with called *names* causes loss of information for humans.
6. **Combining LVTI and programming to the interface technique**: Write a program that exemplifies the usage of var via the *programming to the interface* technique.
7. **Combining LVTI and the diamond operator**: Write a program that exemplifies the usage of var with the *diamond* operator.
8. **Assigning an array to** var: Write a program that assigns an array to var.
9. **Using LVTI in compound declarations**: Explain and exemplify the usage of LVTI with compound declarations.
10. **LVTI and variable scope**: Explain and exemplify why LVTI should minimize the variable's scope as much as possible.
11. **LVTI and the ternary operator**: Write several snippets of code that exemplify the advantages of combining LVTI and the *ternary* operator.
12. **LVTI and** for **loops**: Write several examples that exemplify the usage of LVTI in for loops.
13. **LVTI and streams**: Write several snippets of code that exemplify the usage of LVTI and Java streams.
14. **Using LVTI to break up nested/large chains of expressions**: Write a program that exemplifies the usage of LVTI for breaking up a nested/large chain of expressions.
15. **LVTI and the method return and argument types**: Write several snippets of code that exemplify the usage of LVTI and Java methods in terms of return and argument types.
16. **LVTI and anonymous classes**: Write several snippets of code that exemplify the usage of LVTI in anonymous classes.
17. **LVTI can be final and *effectively final***: Write several snippets of code that exemplify how LVTI can be used for final and *effectively final* variables.
18. **LVTI and lambdas**: Explain via several snippets of code how LVTI can be used in combination with lambda expressions.
19. **LVTI and null *initializers*, instance variables, and catch blocks variables**: Explain with examples how LVTI can be used in combination with null *initializers*, instance variables, and catch blocks.
20. **LVTI and generic types, T**: Write several snippets of code that exemplify how LVTI can be used in combination with generic types.
21. **LVTI, wildcards, covariants, and contravariants**: Write several snippets of code that exemplify how LVTI can be used in combination with wildcards, covariants, and contravariants.

Solutions

The following sections describe the solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations shown here include only the most interesting and important details that are needed to solve the problems. You can download the example solutions to view additional details and experiment with the programs from <https://github.com/PacktPublishing/Java-Coding-Problems>.

78. Simple var example

Starting with version 10, Java comes with JEP 286, or Java LVTI, also known as the var type.

The var identifier is not a Java *keyword*, it is a *reserved type name*.

This is a 100% compile feature with no side effects in terms of bytecode, runtime, or performance. In a nutshell, LVTI is applied to local variables and works as follows: the compiler checks the right-hand side and infers the real type (if the right-hand side is an *initializer*, then it uses that type).

This feature ensures compile-time safety. This means that we cannot compile an application that tries to achieve a wrong assignment. If the compiler has inferred the concrete/actual type of var, we can only assign the values of that type.

There are multiple benefits of LVTI; for example, it reduces code verbosity and mitigates redundancy and *boilerplate* code. Moreover, the time spent to write code can be reduced by LVTI, especially in cases that involve heavy declarations, as follows:

// without var  
Map<Boolean, List<Integer>> evenAndOddMap...  
  
// with var  
var evenAndOddMap = ...Copy

A controversial benefit is represented by code readability. Some voices sustain that using var reduces code readability, while other voices support the opposite. Depending on the use case, it may require a trade-off in readability, but the truth is that, typically, we pay a lot of attention to meaningful names for fields (instance variables) and we neglect the names of local variables. For example, let's consider the following method:

public Object fetchTransferableData(String data)  
 throws UnsupportedFlavorException, IOException {  
  
 StringSelection ss = new StringSelection(data);  
 DataFlavor[] df = ss.getTransferDataFlavors();  
 Object obj = ss.getTransferData(df[0]);  
  
 return obj;  
}Copy

This is a short method; it has a meaningful name and a clean implementation. But checkout the local variables' names. Their names are drastically reduced (they are just shortcuts), but this is not a problem since the left-hand side provides enough information that we can easily understand the type of each local variable. Now, let's write this code using LVTI:

public Object fetchTransferableData(String data)  
 throws UnsupportedFlavorException, IOException {  
  
 var ss = new StringSelection(data);  
 var df = ss.getTransferDataFlavors();  
 var obj = ss.getTransferData(df[0]);  
  
 return obj;  
}Copy

Obviously, the code's readability has decreased since it's now harder to infer the type of the local variables. As the following screenshot reveals, the compiler doesn't have a problem with inferring the correct types, but for humans, this is a lot more difficult:

Texto

Descripción generada automáticamente

The solution to this problem consists of providing a meaningful name to local variables when relying on LVTI. For example, the code can regain its readability if the local variables' names are provided, as follows:

public Object fetchTransferableData(String data)  
 throws UnsupportedFlavorException, IOException {  
  
 var stringSelection = new StringSelection(data);  
 var dataFlavorsArray = stringSelection.getTransferDataFlavors();  
 var obj = stringSelection.getTransferData(dataFlavorsArray[0]);  
  
 return obj;  
}Copy

Nevertheless, the readability problem is also caused by the fact that, typically, we tend to look at the type as primary information and the variable name as secondary information, while this should be the opposite.

Let's look at two more examples that are meant to enforce the aforementioned statements. A method that uses collections (for example, List) is as follows:

// Avoid  
public List<Player> fetchPlayersByTournament(String tournament) {  
  
 var t = tournamentRepository.findByName(tournament);  
 var p = t.getPlayers();  
  
 return p;  
}  
  
// Prefer  
public List<Player> fetchPlayersByTournament(String tournament) {  
  
 var tournamentName = tournamentRepository.findByName(tournament);  
 var playerList = tournamentName.getPlayers();  
  
 return playerList;  
}Copy

Providing meaningful names for local variables doesn't mean falling into the *over-naming* technique.

For example, avoid naming variables by simply repeating the type name:

// Avoid  
var fileCacheImageOutputStream​   
 = new FileCacheImageOutputStream​(..., ...);  
  
// Prefer  
var outputStream​ = new FileCacheImageOutputStream​(..., ...);  
  
// Or  
var outputStreamOfFoo​ = new FileCacheImageOutputStream​(..., ...);Copy

79. Using var with primitive types

The problem of using LVTI with primitive types (int, long, float, and double) is that the expected and inferred types may differ. Obviously, this causes confusion and unexpected behavior in code.

The guilty party in this situation is the *implicit type casting* used by the var type.

For example, let's consider the following two declarations that rely on explicit primitive types:

boolean valid = true; // this is of type boolean  
char c = 'c'; // this is of type charCopy

Now, let's replace the explicit primitive type with LVTI:

var valid = true; // inferred as boolean  
var c = 'c'; // inferred as charCopy

Nice! There are no problems so far! Now, let's have a look at another set of declarations based on explicit primitive types:

int intNumber = 10; // this is of type int  
long longNumber = 10; // this is of type long  
float floatNumber = 10; // this is of type float, 10.0  
double doubleNumber = 10; // this is of type double, 10.0Copy

Let's follow the logic from the first example and replace the explicit primitive types with LVTI:

// Avoid  
var intNumber = 10; // inferred as int  
var longNumber = 10; // inferred as int  
var floatNumber = 10; // inferred as int  
var doubleNumber = 10; // inferred as intCopy

Conforming to the following screenshot, all four variables have been inferred as integers:

Texto

Descripción generada automáticamente

The solution to this problem consists of using explicit Java *literals*:

// Prefer  
var intNumber = 10; // inferred as int  
var longNumber = 10L; // inferred as long  
var floatNumber = 10F; // inferred as float, 10.0  
var doubleNumber = 10D; // inferred as double, 10.0Copy

Finally, let's consider the case of a number with decimals, as follows:

var floatNumber = 10.5; // inferred as doubleCopy

The variable name suggests that 10.5 is float, but actually, it is inferred as double. So, it is advisable to rely on *literals* even for numbers with decimals (especially for numbers of the float type):

var floatNumber = 10.5F; // inferred as floatCopy

80. Using var and implicit type casting to sustain the code's maintainability

In the previous section, *Using var with primitive types*, we saw that combining var with *implicit type casting* can cause real problems. But in certain scenarios, this combination can be advantageous and sustain the code's maintainability.

Let's consider the following scenario—we need to write a method that sits between two existing methods of an external API named ShoppingAddicted (by extrapolation, these methods can be two web services, endpoints, and so on). One method is dedicated to returning the best price for a given shopping cart. Basically, this method takes a bunch of products and queries different online stores to fetch the best price.

The resulting price is returned as int. A stub of this method is listed as follows:

public static int fetchBestPrice(String[] products) {  
  
 float realprice = 399.99F; // code to query the prices in stores  
 int price = (int) realprice;  
  
 return price;  
}Copy

The other method receives the price as int and performs the payment. If the payment is successful, it returns true:

public static boolean debitCard(int amount) {  
  
 return true;  
}Copy

Now, by programming with respect to this code, our method will act as a client, as follows (the customers can decide what items to buy, and our code will return the best price for them and debit their cards accordingly):

// Avoid  
public static boolean purchaseCart(long customerId) {  
  
 int price = ShoppingAddicted.fetchBestPrice(new String[0]);  
 boolean paid = ShoppingAddicted.debitCard(price);  
  
 return paid;  
}Copy

But after some time, the owners of the ShoppingAddicted API realize that they lose money by converting the real price into int (for example, the real price is 399.99, but in int form, it's 399.0, which means a loss of 99 cents). So, they decide to quit this practice and return the real price as float:

public static float fetchBestPrice(String[] products) {  
  
 float realprice = 399.99F; // code to query the prices in stores  
  
 return realprice;  
}Copy

Since the returned price is float, debitCard() is updated as well:

public static boolean debitCard(float amount) {  
  
 return true;  
}Copy

But once we upgrade to the new release of the ShoppingAddicted API, the code will fail with a possible *lossy conversion from* float into int exceptions. This is normal since our code expects int. Since our code doesn't tolerate these modifications well, the code needs to be modified accordingly.

Nevertheless, if we have anticipated this situation and used var instead of int, then the code will work without problems thanks to *implicit type casting*:

// Prefer  
public static boolean purchaseCart(long customerId) {  
  
 var price = ShoppingAddicted.fetchBestPrice(new String[0]);  
 var paid = ShoppingAddicted.debitCard(price);  
  
 return paid;  
}Copy

81. Explicit downcast or better avoid var

In the *Using var with primitive types* section, we talked about using *literals* with primitive types (int, long, float, and double) to avoid issues caused by *implicit type casting.* But not all Java primitive types can take advantage of *literals*. In such a situation, the best approach is to avoid using var. But let's see why!

Check out the following declarations of byte and short variables:

byte byteNumber = 25; // this is of type byte  
short shortNumber = 1463; // this is of type shortCopy

If we replace the explicit types with var, then the inferred type will be int:

var byteNumber = 25; // inferred as int  
var shortNumber = 1463; // inferred as intCopy

Unfortunately, there are no *literals* available for these two primitive types. The only approach to help the compiler to infer the correct types is to rely on an explicit downcast:

var byteNumber = (byte) 25; // inferred as byte  
var shortNumber = (short) 1463; // inferred as shortCopy

While this code compiles successfully and works as expected, we cannot say that using var brought any value compared to using explicit types. So, in this case, it is better to avoid var and explicit downcast.

82. Avoid using var if the called names don't contain enough type information for humans

Well, var is not a silver bullet, and this problem will highlight this once again. The following snippet of code can be written using explicit types or var without losing information:

// using explicit types  
MemoryCacheImageInputStream is =  
 new MemoryCacheImageInputStream(...);  
JavaCompiler jc = ToolProvider.getSystemJavaCompiler();  
StandardJavaFileManager fm = compiler.getStandardFileManager(...);Copy

So, migrating the preceding snippet of code to var will result in the following code (the variables names have been chosen by visually inspecting the called *names* from the right-hand side):

// using var  
var inputStream = new MemoryCacheImageInputStream(...);  
var compiler = ToolProvider.getSystemJavaCompiler();  
var fileManager = compiler.getStandardFileManager(...);Copy

The same will happen at the border of over-naming:

// using var  
var inputStreamOfCachedImages = new MemoryCacheImageInputStream(...);  
var javaCompiler = ToolProvider.getSystemJavaCompiler();  
var standardFileManager = compiler.getStandardFileManager(...);Copy

So, the preceding code doesn't raise any issues in choosing the variable's names and readability. The called *names* contain enough information for humans to feel comfortable with var.

But let's consider the following snippet of code:

// Avoid  
public File fetchBinContent() {  
 return new File(...);  
}  
  
// called from another place  
// notice the variable name, bin  
var bin = fetchBinContent();Copy

For humans, it is pretty difficult to infer the type that's returned by the called *name* without inspecting the returned type of this *name*, fetchBinContent(). As a rule of thumb, in such cases, the solution should avoid var and rely on explicit types since there is not enough information on the right-hand side for us to choose a proper name for the variable and obtain highly readable code:

// called from another place  
// now the left-hand side contains enough information  
File bin = fetchBinContent();Copy

So, if var in combination with the called *names* causes loss of clarity, then it is better to avoid the usage of var. Ignoring this statement may lead to confusion and will increase the time needed to understand and/or extend the code.

Consider another example based on the java.nio.channels.Selector class. This class exposes a static method named open() that returns a newly opened Selector. But if we capture this return in a variable declared with var, it's tempting to think that this method may return a boolean representing the success of opening the current selector. Using var without considering the possible loss of clarity can produce exactly these kinds of problems. Just a few issues like this one and the code will become a real pain.

83. Combining LVTI and programming to the interface technique

Java best practices encourage us to bind the code to the abstraction. In other words, we need to rely on the *programming to the interface* technique.

This technique fits very well for collection declarations. For example, it is advisable to declare ArrayList as follows:

List<String> players = new ArrayList<>();Copy

We should also avoid something like this:

ArrayList<String> players = new ArrayList<>();Copy

By following the first example, the code instantiates the ArrayList class (or HashSet, HashMap, and so on), but declares a variable of the List type (or Set, Map, and so on). Since List, Set, Map, and many more are interfaces (or contracts), it is very easy to replace the instantiation with other implementation of List (Set, and Map) without subsequent modifications being made to the code.

Unfortunately, LVTI cannot take advantage of the *programming to the interface* technique. In other words, when we use var, the inferred type is the concrete implementation, not the contract. For example, replacing List<String> with var will result in the inferred type, ArrayList<String>:

// inferred as ArrayList<String>  
var playerList = new ArrayList<String>();Copy

Nevertheless, there are some explanations that sustain this behavior:

* LVTI acts at the local level (local variables) where the *programming to the interface* technique is used less than method parameters/return types or field types.
* Since local variables have a small scope, the modifications that are induced by switching to another implementation should be small as well. Switching implementation should have a small impact on detecting and fixing the code.
* LVTI sees the code from the right-hand side as an *initializer* that's useful for inferring the actual type. If this *initializer* is going to be modified in the future, then the inferred type may differ, and this will cause problems in the code that uses this variable.

84. Combining LVTI and the diamond operator

As a rule of thumb, LVTI combined with the *diamond* operator may result in unexpected inferred types if the information that's needed for inferring the expected type is not present in the right-hand side.

Before JDK 7, that is, Project Coin, List<String> would be declared as follows:

List<String> players = new ArrayList<String>();Copy

Basically, the preceding example explicitly specifies the generic class's instantiation parameter type. Starting with JDK 7, Project Coin introduced the *diamond* operator, which is capable of inferring the generic class instantiation parameter type, as follows:

List<String> players = new ArrayList<>();Copy

Now, if we think about this example in terms of LVTI, we will get the following result:

var playerList = new ArrayList<>();Copy

But what will be the inferred type now? Well, we have a problem because the inferred type will be ArrayList<Object>, not ArrayList<String>. The explanation is quite obvious: the information that's needed for inferring the expected type (String) is not present (notice that there is no String type mentioned in the right-hand side). This instructs LVTI to infer the type that is the broadest applicable type, which, in this case, is Object.

But if ArrayList<Object> was not our intention, then we need a solution to this problem. The solution is to provide the information that's needed for inferring the expected type, as follows:

var playerList = new ArrayList<String>();Copy

Now, the inferred type is ArrayList<String>. The type can be inferred indirectly as well. See the following example:

var playerStack = new ArrayDeque<String>();  
  
// inferred as ArrayList<String>  
var playerList = new ArrayList<>(playerStack);Copy

It can also be inferred indirectly in the following way:

Player p1 = new Player();  
Player p2 = new Player();  
var listOfPlayer = List.of(p1, p2); // inferred as List<Player>  
  
// Don't do this!  
var listOfPlayer = new ArrayList<>(); // inferred as ArrayList<Object>  
listOfPlayer.add(p1);  
listOfPlayer.add(p2);Copy

85. Assigning an array to var

As a rule of thumb, assigning an array to var doesn't require brackets, []. Defining an array of int via the corresponding explicit type can be done as follows:

int[] numbers = new int[10];  
  
// or, less preferred  
int numbers[] = new int[10];Copy

Now, trying to intuit how to use var instead of int may result in the following attempts:

var[] numberArray = new int[10];  
var numberArray[] = new int[10];Copy

Unfortunately, none of these two approaches will compile. The solution requires us to remove the brackets from the left-hand side:

// Prefer  
var numberArray = new int[10]; // inferred as array of int, int[]  
numberArray[0] = 3; // works  
numberArray[0] = 3.2; // doesn't work  
numbers[0] = "3"; // doesn't workCopy

There is a common practice to initialize an array at declaration time, as follows:

// explicit type work as expected  
int[] numbers = {1, 2, 3};Copy

However, trying to use var will not work (will not compile):

// Does not compile  
var numberArray = {1, 2, 3};  
var numberArray[] = {1, 2, 3};  
var[] numberArray = {1, 2, 3};Copy

This code doesn't compile because the right-hand side doesn't have its own type.

86. Using LVTI in compound declarations

A compound declaration allows us to declare a group of variables of the same type without repeating the type. The type is specified a single time and the variables are demarcated by a comma:

// using explicit type  
String pending = "pending", processed = "processed",   
 deleted = "deleted";Copy

Replacing String with var will result in code that doesn't compile:

// Does not compile  
var pending = "pending", processed = "processed", deleted = "deleted";Copy

The solution to this problem is to transform the compound declaration into one declaration per single line:

// using var, the inferred type is String  
var pending = "pending";  
var processed = "processed";  
var deleted = "deleted";Copy

So, as a rule of thumb, LVTI cannot be used in compound declarations.

87. LVTI and variable scope

The clean code best practices include keeping a small scope for all local variables. This is one of the clean code golden rules that was followed even before the existence of LVTI.

This rule sustains the readability and debugging phase. It can speed up the process of finding bugs and writing fixes. Consider the following example that breaks down this rule:

// Avoid  
...  
var stack = new Stack<String>();  
stack.push("John");  
stack.push("Martin");  
stack.push("Anghel");  
stack.push("Christian");  
  
// 50 lines of code that doesn't use stack  
  
// John, Martin, Anghel, Christian  
stack.forEach(...);Copy

So, the preceding code declares a stack with four names, contains 50 lines of code that don't use this stack, and finishes with a loop of this stack via the forEach() method. This method is inherited from java.util.Vector and will loop the stack as any vector (John, Martin, Anghel, Christian). This is the order of traversal that we want.

But later on, we decide to switch from the stack to ArrayDeque (the reason is irrelevant). This time, the forEach() method will be the one provided by the ArrayDeque class. The behavior of this method is different from Vector.forEach(), meaning that the loop will traverse the entries following the **Last In First Out** (**LIFO**) traversal (Christian, Anghel, Martin, John):

// Avoid  
...  
var stack = new ArrayDeque<String>();  
stack.push("John");  
stack.push("Martin");  
stack.push("Anghel");  
stack.push("Christian");  
  
// 50 lines of code that doesn't use stack  
  
// Christian, Anghel, Martin, John  
stack.forEach(...);Copy

This was not our intention! We switched to ArrayDeque for other purposes, not for affecting the looping order. But it is pretty difficult to see that there was a bug in the code since the part of the code containing the forEach() part is not in proximity of the code where we completed the modifications (50 lines below this line of code). It is our duty to come up with a solution that maximizes the chances of getting this bug fixed quickly and avoiding a bunch of scrolling up and down to understand what is going on. The solution consists of following the clean code rule we invoked earlier and writing this code with a small scope for the stack variable:

// Prefer  
...  
var stack = new Stack<String>();  
stack.push("John");  
stack.push("Martin");  
stack.push("Anghel");  
stack.push("Christian");  
  
// John, Martin, Anghel, Christian  
stack.forEach(...);  
  
// 50 lines of code that doesn't use stackCopy

Now, when we switch from Stack to ArrayQueue, we should notice the bug faster and be able to fix it.

88. LVTI and the ternary operator

As long as it is written correctly, the *ternary* operator allows us to use different types of operands on the right-hand side. For example, the following code will not compile:

// Does not compile  
List evensOrOdds = containsEven ?  
 List.of(10, 2, 12) : Set.of(13, 1, 11);  
  
// Does not compile  
Set evensOrOdds = containsEven ?  
 List.of(10, 2, 12) : Set.of(13, 1, 11);Copy

Nevertheless, this code can be fixed by rewriting it using the correct/supported explicit types:

Collection evensOrOdds = containsEven ?  
 List.of(10, 2, 12) : Set.of(13, 1, 11);  
  
Object evensOrOdds = containsEven ?  
 List.of(10, 2, 12) : Set.of(13, 1, 11);Copy

A similar attempt will fail for the following snippet of code:

// Does not compile  
int numberOrText = intOrString ? 2234 : "2234";  
  
// Does not compile  
String numberOrText = intOrString ? 2234 : "2234";Copy

However, it can be fixed like this:

Serializable numberOrText = intOrString ? 2234 : "2234";  
  
Object numberOrText = intOrString ? 2234 : "2234";Copy

So, in order to have a *ternary* operator with different types of operands on the right-hand side, the developer must match the correct type that supports both conditional branches. Alternatively, the developer can rely on LVTI, as follows (of course, this works for the same types of operands as well):

// inferred type, Collection<Integer>  
var evensOrOddsCollection = containsEven ?  
 List.of(10, 2, 12) : Set.of(13, 1, 11);  
  
// inferred type, Serializable  
var numberOrText = intOrString ? 2234 : "2234";Copy

Don't conclude from these examples that the var type is inferred at runtime! It is NOT!

89. LVTI and for loops

Declaring simple for loops using explicit types is a trivial task, as follows:

// explicit type  
for (int i = 0; i < 5; i++) {  
 ...  
}Copy

Alternatively, we can use an enhanced for loop:

List<Player> players = List.of(  
 new Player(), new Player(), new Player());  
for (Player player: players) {  
 ...  
}Copy

Starting with JDK 10, we can replace the explicit types of the variables, i and player, with var, as follows:

for (var i = 0; i < 5; i++) { // i is inferred of type int  
 ...  
}  
  
for (var player: players) { // i is inferred of type Player  
 ...  
}Copy

Using var can be helpful when the type of a looped array, collection, and so on is changed. For example, by using var, both versions of the following array can be looped without specifying the explicit type:

// a variable 'array' representing an int[]  
int[] array = { 1, 2, 3 };  
  
// or the same variable, 'array', but representing a String[]  
String[] array = {  
 "1", "2", "3"  
};  
  
// depending on how 'array' is defined   
// 'i' will be inferred as int or as String  
for (var i: array) {  
 System.out.println(i);  
}Copy

90. LVTI and streams

Let's consider the following Stream<Integer> stream:

// explicit type  
Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);  
numbers.filter(t -> t % 2 == 0).forEach(System.out::println);Copy

Using LVTI instead of Stream<Integer> is pretty straightforward. Simply replace Stream<Integer> with var, as follows:

// using var, inferred as Stream<Integer>  
var numberStream = Stream.of(1, 2, 3, 4, 5);  
numberStream.filter(t -> t % 2 == 0).forEach(System.out::println);Copy

Here is another example:

// explicit types  
Stream<String> paths = Files.lines(Path.of("..."));  
List<File> files = paths.map(p -> new File(p)).collect(toList());  
  
// using var  
// inferred as Stream<String>  
var pathStream = Files.lines(Path.of(""));  
  
// inferred as List<File>  
var fileList = pathStream.map(p -> new File(p)).collect(toList());Copy

It looks like Java 10, LVTI, Java 8, and the Stream API make a good team.

91. Using LVTI to break up nested/large chains of expressions

Large/nested expressions are usually snippets of codes that look pretty impressive and are intimidating. They are commonly seen as pieces of *smart* or *clever* code. It is controversial as to whether this is good or bad, but most likely, the balance tends to be in favor of those who claim that such code should be avoided. For example, check out the following expression:

List<Integer> ints = List.of(1, 1, 2, 3, 4, 4, 6, 2, 1, 5, 4, 5);  
  
// Avoid  
int result = ints.stream()  
 .collect(Collectors.partitioningBy(i -> i % 2 == 0))  
 .values()  
 .stream()  
 .max(Comparator.comparing(List::size))  
 .orElse(Collections.emptyList())  
 .stream()  
 .mapToInt(Integer::intValue)  
 .sum();Copy

Such expressions can be written deliberately or they can represent the final result of an incremental process that enriches an initially small expression in time. Nevertheless, when such expressions start to become gaps in readability, they must be broken into pieces via local variables. But this is not fun and can be considered exhausting work that we want to avoid:

List<Integer> ints = List.of(1, 1, 2, 3, 4, 4, 6, 2, 1, 5, 4, 5);  
  
// Prefer  
Collection<List<Integer>> evenAndOdd = ints.stream()  
 .collect(Collectors.partitioningBy(i -> i % 2 == 0))  
 .values();  
  
List<Integer> evenOrOdd = evenAndOdd.stream()  
 .max(Comparator.comparing(List::size))  
 .orElse(Collections.emptyList());  
  
int sumEvenOrOdd = evenOrOdd.stream()  
 .mapToInt(Integer::intValue)  
 .sum();Copy

Check out the types of the local variables in the preceding code. We have Collection<List<Integer>>, List<Integer>, and int. It is obvious that these explicit types require some time to be fetched and written. This may be a good reason to avoid breaking this expression into pieces. Nevertheless, the triviality of using the var type instead of explicit types is tempting if we wish to adopt the local variable's style because it saves time that's usually spent fetching the explicit types:

var intList = List.of(1, 1, 2, 3, 4, 4, 6, 2, 1, 5, 4, 5);  
  
// Prefer  
var evenAndOdd = intList.stream()  
 .collect(Collectors.partitioningBy(i -> i % 2 == 0))  
 .values();  
  
var evenOrOdd = evenAndOdd.stream()  
 .max(Comparator.comparing(List::size))  
 .orElse(Collections.emptyList());  
  
var sumEvenOrOdd = evenOrOdd.stream()  
 .mapToInt(Integer::intValue)  
 .sum();Copy

Awesome! Now, it is the compiler's job to infer the types of these local variables. We only choose the points where we break the expression and demarcate them with var.

92. LVTI and the method return and argument types

As a rule of thumb, LVTI cannot be used as a return method type or as an argument method type; instead, variables of the var type can be passed as method arguments or store a return method. Let's iterate these statements via several examples:

* LVTI cannot be used as the method return type—the following code doesn't compile:

// Does not compile  
public var fetchReport(Player player, Date timestamp) {  
  
 return new Report();  
}Copy

* LVTI cannot be used as a method argument type—the following code doesn't compile:

public Report fetchReport(var player, var timestamp) {  
  
 return new Report();  
}Copy

* Variables of the var type can be passed as method arguments or store a return method—the following code compiles successfully and it works:

public Report checkPlayer() {  
  
 var player = new Player();  
 var timestamp = new Date();  
 var report = fetchReport(player, timestamp);  
  
 return report;  
}  
  
public Report fetchReport(Player player, Date timestamp) {  
  
 return new Report();  
}Copy

93. LVTI and anonymous classes

LVTI can be used for anonymous classes. Let's take a look at the following example of an anonymous class that uses an explicit type for the weighter variable:

public interface Weighter {  
 int getWeight(Player player);  
}  
  
Weighter weighter = new Weighter() {  
 @Override  
 public int getWeight(Player player) {  
 return ...;  
 }  
};  
  
Player player = ...;  
int weight = weighter.getWeight(player);Copy

Now, look at what happens if we use LVTI:

var weighter = new Weighter() {  
 @Override  
 public int getWeight(Player player) {  
 return ...;  
 }  
};Copy

94. LVTI can be final and effectively final

As a quick reminder, *starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or effectively final. A variable or parameter whose value is never changed after it is initialized is effectively final.*

The following snippet of code represents the use case of an *effectively final* variable (trying to reassign the ratio variable will result in an error, which means that this variable is *effectively final*) and two final variables (trying to reassign the limit and bmi variables will result in an error, which means that these variables are final):

public interface Weighter {  
 float getMarginOfError();  
}  
  
**float ratio = fetchRatio();** // this is effectively final  
  
var weighter = new Weighter() {  
 @Override  
 public float getMarginOfError() {  
 return ratio \* ...;  
 }  
};  
  
**ratio = fetchRatio();** // this reassignment will cause error  
  
public float fetchRatio() {  
  
 **final float limit = new Random().nextFloat();** // this is final  
 **final float bmi = 0.00023f;**  // this is final  
  
 **limit = 0.002f;** // this reassignment will cause error  
 **bmi = 0.25f;** // this reassignment will cause error  
  
 return limit \* bmi / 100.12f;  
}Copy

Now, let's replace the explicit types with var. The compiler will infer the correct types for these variables (ratio, limit, and bmi) and maintain their state—ratio will be *effectively final* while limit and bmi are final. Trying to reassign any of them will cause a specific error:

**var ratio = fetchRatio(); // this is effectively final**  
var weighter = new Weighter() {  
 @Override  
 public float getMarginOfError() {  
 return ratio \* ...;  
 }  
};  
  
**ratio = fetchRatio(); // this reassignment will cause error**  
public float fetchRatio() {  
  
 **final var limit = new Random().nextFloat();** // this is final  
 **final var bmi = 0.00023f;** // this is final  
 **limit = 0.002f;** // this reassignment will cause error  
 **bmi = 0.25f;** // this reassignment will cause error  
 return limit \* bmi / 100.12f;  
}Copy

95. LVTI and lambdas

The problem with using LVTI and lambdas is that the concrete type cannot be inferred. Lambdas and method reference *initializers* are not allowed. This statement is part of var limitations; therefore, lambda expressions and method references need explicit target types.

For example, the following snippet of code will not compile:

// Does not compile  
// lambda expression needs an explicit target-type  
var incrementX = x -> x + 1;  
  
// method reference needs an explicit target-type  
var exceptionIAE = IllegalArgumentException::new;Copy

Since var cannot be used, these two snippets of code need to be written as follows:

Function<Integer, Integer> incrementX = x -> x + 1;  
Supplier<IllegalArgumentException> exceptionIAE   
 = IllegalArgumentException::new;Copy

But in the context of lambdas, Java 11 allows us to use var in lambda parameters. For example, the following code is working in Java 11 (more details can be found in *JEP 323: Local-Variable Syntax for Lambda Parameters* at <https://openjdk.java.net/jeps/323>):

@FunctionalInterface  
public interface Square {  
 int calculate(int x);  
}  
  
Square square = (var x) -> x \* x;Copy

However, keep in mind that the following will not work:

var square = (var x) -> x \* x; // cannot inferCopy

96. LVTI and null initializers, instance variables, and catch blocks variables

What does LVTI have in common with null *initializers*, instance variables, and catch blocks variables? Well, LVTI cannot be used with any of them. The following attempts will fail:

* LVTI cannot be used with null *initializers*:

// result in an error of type: variable initializer is 'null'  
var message = null;  
  
// result in: cannot use 'var' on variable without initializer  
var message;Copy

* LVTI cannot be used with instance variables (fields):

public class Player {  
  
 private var age; // error: 'var' is not allowed here  
 private var name; // error: 'var' is not allowed here  
 ...  
}Copy

* LVTI cannot be used in catch block variables:

try {  
 TimeUnit.NANOSECONDS.sleep(1000);  
} catch (var ex) { ... }Copy

Try-with-resource

On the other hand, the var type is a very nice fit for *try-with-resource*, as in the following example:

// explicit type  
try (PrintWriter writer = new PrintWriter(new File("welcome.txt"))) {  
 writer.println("Welcome message");  
}

// using var  
try (var writer = new PrintWriter(new File("welcome.txt"))) {  
 writer.println("Welcome message");  
}Copy

97. LVTI and generic types, T

In order to understand how LVTI can be combined with generic types, let's start with an example. The following method is a classical usage case of a generic type, T:

public static <T extends Number> T add(T t) {  
 T temp = t;  
 ...  
 return temp;  
}Copy

In this case, we can replace T with var and the code will work fine:

public static <T extends Number> T add(T t) {  
 var temp = t;  
 ...  
 return temp;  
}Copy

So, local variables that have generic types can take advantage of LVTI. Let's look at some other examples, first using the generic type, T:

public <T extends Number> T add(T t) {  
  
 List<T> numberList = new ArrayList<T>();  
 numberList.add(t);  
 numberList.add((T) Integer.valueOf(3));  
 numberList.add((T) Double.valueOf(3.9));  
  
 // error: incompatible types: String cannot be converted to T  
 // numbers.add("5");  
  
 return numberList.get(0);  
}Copy

Now, let's replace List<T> with var:

public <T extends Number> T add(T t) {  
  
 var numberList = new ArrayList<T>();  
 numberList.add(t);  
 numberList.add((T) Integer.valueOf(3));  
 numberList.add((T) Double.valueOf(3.9));  
  
 // error: incompatible types: String cannot be converted to T  
 // numbers.add("5");  
  
 return numberList.get(0);  
}Copy

Pay attention and double-check the ArrayList instantiation for the presence of T. Don't do this (this will be inferred as ArrayList<Object> and will ignore the real type behind the generic type, T):

var numberList = new ArrayList<>();Copy

98. LVTI, wildcards, covariants, and contravariants

Replacing wildcards, covariants, and contravariants with LVTI is a delicate job and should be done with full awareness of the consequences.

LVTI and wildcards

First, let's talk about LVTI and wildcards (?). It is a common practice to associate wildcards with Class and write something like this:

// explicit type  
Class<?> clazz = Long.class;Copy

In such cases, there is no problem with using var instead of Class<?>. Depending on the right-hand side type, the compiler will infer the correct type. In this example, the compiler will infer Class<Long>.

But notice that replacing wildcards with LVTI should be done carefully and that you should be aware of the consequences (or side effects). Let's look at an example where replacing a wildcard with var is a bad choice. Consider the following piece of code:

Collection<?> stuff = new ArrayList<>();  
stuff.add("hello"); // compile time error  
stuff.add("world"); // compile time errorCopy

This code doesn't compile because of incompatible types. A very bad approach would be to fix this code by replacing the wildcard with var, as follows:

var stuff = new ArrayList<>();  
strings.add("hello"); // no error  
strings.add("world"); // no errorCopy

By using var, the error will disappear, but this is not what we had in mind when we wrote the preceding code (the code with type incompatibility errors). So, as a rule of thumb, don't replace *Foo*<?> with var just because some annoying errors will disappear by magic! Try to think about what the intended task was and act accordingly. For example, maybe in the preceding snippet of code, we tried to define ArrayList<String> and, by mistake, ended up with Collection<?>.

LVTI and covariants/contravariants

Replacing covariants (*Foo*<? extends T>) or contravariants (*Foo*<? super T>) with LVTI is a dangerous approach and should be avoided.

Check out the following snippet of code:

// explicit types  
Class<? extends Number> intNumber = Integer.class;  
Class<? super FilterReader> fileReader = Reader.class;Copy

In the covariant, we have an upper bound represented by the Number class, while in the contravariant, we have a lower bound represented by the FilterReader class. Having these bounds (or constraints) in place, the following code will trigger a specific compile-time error:

// Does not compile  
// error: Class<Reader> cannot be converted   
// to Class<? extends Number>  
Class<? extends Number> intNumber = Reader.class;  
  
// error: Class<Integer> cannot be converted   
// to Class<? super FilterReader>  
Class<? super FilterReader> fileReader = Integer.class;Copy

Now, let's use var instead of the preceding covariant and contravariant:

// using var  
var intNumber = Integer.class;  
var fileReader = Reader.class;Copy

This code will not cause any issues. Now, we can assign any class to these variables so that our bounds/constraints vanish. This is not what we intended to do:

// this will compile just fine  
var intNumber = Reader.class;  
var fileReader = Integer.class;Copy

So, using var in place of our covariant and contravariant was a bad choice!

Summary

This was the last problem of this chapter. Take a look at *JEP 323: Local-Variable Syntax for Lambda Parameters* (<https://openjdk.java.net/jeps/323>) and *JEP 301: Enhanced* *Enums* (<http://openjdk.java.net/jeps/301>) for more information. Adopting these features should be pretty smooth as long as you are familiar with the problems that were covered in this chapter.

Download the applications from this chapter to see the results and additional details.

Arrays, Collections, and Data Structures

This chapter includes 30 problems that involve arrays, collections, and several data structures. The aim is to provide solutions to a category of problems encountered in a wide range of applications, including sorting, finding, comparing, ordering, reversing, filling, merging, copying, and replacing. The solutions provided are implemented in Java 8-12 and they can also be used as a basis for solving other related issues. At the end of this chapter, you will have at your disposal a solid breadth of knowledge that will prove useful in solving a variety of problems involving arrays, collections, and data structures.

Problems

Use the following problems to test your programming prowess based on arrays, collections, and data structures. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Sorting an array**: Write several programs that exemplify different sorting algorithms for arrays. Also, write a program for shuffling arrays.
2. **Finding an element in an array**: Write several programs that exemplify how to find the given element (primitive and object) in a given array. Find the index and/or simply check whether the value is in the array.
3. **Checking whether two arrays are equal or mismatches**: Write a program that checks whether the two given arrays are equals or whether there is a mismatch.
4. **Comparing two arrays lexicographically**: Write a program that compares the given arrays lexicographically.
5. **Creating a stream from an array**: Write a program that creates a stream from the given array.
6. **Minimum, maximum, and average of an array**: Write a program that computes the maximum, minimum, and average of the given array.
7. **Reversing an array**: Write a program that reverses the given array.
8. **Filling and setting an array**: Write several examples for filling up an array and setting all elements based on a generator function to compute each element.
9. **Next Greater Element** (**NGE**): Write a program that returns the NGE for each element of an array.
10. **Changing array size**: Write a program that adds an element to an array by increasing its size by one. In addition, write a program that increases the size of an array with the given length.
11. **Creating unmodifiable/immutable collections**: Write several examples that create unmodifiable and immutable collections.
12. **Mapping a default value**: Write a program that gets a value from Map or a default value.
13. **Computing whether absent/present in a** Map: Write a program that computes the value of an absent key or a new value of a present key.
14. **Removal from a** Map: Write a program that removes from a Map by means of the given key.
15. **Replacing entries from a** Map: Write a program that replaces the given entries from a Map.
16. **Comparing two maps**: Write a program that compares two maps.
17. **Merging two maps**: Write a program that merges two given maps.
18. **Copying** HashMap: Write a program that performs a shallow and deep copy of HashMap.
19. **Sorting a** Map: Write a program that sorts a Map.
20. **Removing all elements of a collection that match a predicate**: Write a program that removes all elements of a collection that match the given predicate.
21. **Converting a collection into an array**: Write a program that converts a collection into an array.
22. **Filtering a collection by** List: Write several solutions for filtering a collection by a List. Reveal the best way of doing this.
23. **Replacing elements of a** List: Write a program that replaces each element of a List with the result of applying a given operator to it.
24. **Thread-safe collections, stacks, and queues**: Write several programs that exemplify the usage of Java thread-safe collections.
25. **Breadth-first search** (**BFS**): Write a program that implements the BFS algorithm.
26. **Trie**: Write a program that implements a Trie data structure.
27. **Tuple**: Write a program that implements a Tuple data structure.
28. **Union Find**: Write a program that implements the Union Find algorithm.
29. **Fenwick Tree or Binary Indexed Tree**: Write a program that implements the Fenwick Tree algorithm.
30. **Bloom filter**: Write a program that implements the Bloom filter algorithm.

Solutions

The following sections describe solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations shown here include only the most interesting and important details needed to solve the problems. Download the example solutions to see additional details and to experiment with the programs at <https://github.com/PacktPublishing/Java-Coding-Problems>.

99. Sorting an array

Sorting an array is a common task encountered in a lot of domains/applications. It is so common that Java provides a built-in solution for sorting arrays of primitives and objects using a comparator. This solution works very well and is the preferable way to go in most of the cases. Let's take a look at the different solutions in the next section.

JDK built-in solutions

The built-in solution is named sort() and it comes in many different flavors in the java.util.Arrays class (15+ flavors).

Behind the sort() method, there is a performant sorting algorithm of the Quicksort type, named Dual-Pivot Quicksort.

Let's assume that we need to sort an array of integers by natural order (primitive int). For this, we can rely on Arrays.sort(int[] a), as in the following example:

int[] integers = new int[]{...};  
Arrays.sort(integers);Copy

Sometimes, we need to sort an array of an object. Let's assume that we have a class as Melon:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 public Melon(String type, int weight) {  
 this.type = type;  
 this.weight = weight;  
 }  
  
 // getters omitted for brevity  
}Copy

An array of Melon can be sorted by ascending weight via the proper Comparator:

Melon[] melons = new Melon[] { ... };  
  
Arrays.sort(melons, new Comparator<Melon>() {  
 @Override  
 public int compare(Melon melon1, Melon melon2) {  
 return Integer.compare(melon1.getWeight(), melon2.getWeight());  
 }  
});Copy

The same result can be obtained by rewriting the preceding code via a lambda expression:

Arrays.sort(melons, (Melon melon1, Melon melon2)   
 -> Integer.compare(melon1.getWeight(), melon2.getWeight()));Copy

Moreover, arrays provide a method for sorting elements in parallel, parallelSort(). The sorting algorithm used behind the scenes is a parallel sort-merge based on ForkJoinPool that breaks up the array into sub-arrays that are themselves sorted and then merged. Here is an example:

Arrays.parallelSort(melons, new Comparator<Melon>() {  
 @Override  
 public int compare(Melon melon1, Melon melon2) {  
 return Integer.compare(melon1.getWeight(), melon2.getWeight());  
 }  
});Copy

Or, via a lambda expression, we have the following example:

Arrays.parallelSort(melons, (Melon melon1, Melon melon2)   
 -> Integer.compare(melon1.getWeight(), melon2.getWeight()));Copy

The preceding examples sort an array in ascending order, but sometimes, we need to sort it by descending order. When we sort an array of Object and rely on a Comparator, we can simply multiply the result returned by Integer.compare() by -1:

Arrays.sort(melons, new Comparator<Melon>() {  
 @Override  
 public int compare(Melon melon1, Melon melon2) {  
 return (-1) \* Integer.compare(melon1.getWeight(),   
 melon2.getWeight());  
 }  
});Copy

Or, we can simply switch the arguments in the compare() method.

In the case of an array of boxed primitive types, the solution can rely on the Collections.reverse() method, as in the following example:

Integer[] integers = new Integer[] {3, 1, 5};  
  
// 1, 3, 5  
Arrays.sort(integers);  
  
// 5, 3, 1  
Arrays.sort(integers, Collections.reverseOrder());Copy

Unfortunately, there is no built-in solution for sorting an array of primitives in descending order. Most commonly, if we still want to rely on Arrays.sort(), the solution to this problem consists of reversing the array (O(n)) after it is sorted in ascending order:

// sort ascending  
Arrays.sort(integers);  
  
// reverse array to obtain it in descending order  
for (int leftHead = 0, rightHead = integers.length - 1;  
 leftHead < rightHead; leftHead++, rightHead--) {  
  
 int elem = integers[leftHead];  
 integers[leftHead] = integers[rightHead];  
 integers[rightHead] = elem;  
}Copy

Another solution can rely on Java 8 functional style and boxing (be aware that boxing is a pretty time-consuming operation):

int[] descIntegers = Arrays.stream(integers)  
 .boxed() //or .mapToObj(i -> i)  
 .sorted((i1, i2) -> Integer.compare(i2, i1))  
 .mapToInt(Integer::intValue)  
 .toArray();Copy

Other sorting algorithms

Well, there are plenty of other sorting algorithms out there. Each of them has pros and cons, and the best way to choose between them is to benchmark the situation specific to the application.

Let's examine some of these, as highlighted in the next section, and begin with a pretty slow algorithm.

Bubble sort

Bubble sort is a simple algorithm that basically bubbles up the elements of the array. This means that it traverses the array multiple times and swaps the adjacent elements if they are in the wrong order, as in the following diagram:

Forma, Rectángulo

Descripción generada automáticamente con confianza media

The time complexity cases are as follows: best case O(n), average case O(n2), and worst case O(n2)

The space complexity case is as follows: worst case O(1)

A utility method implementing the Bubble sort is as follows:

public static void bubbleSort(int[] arr) {  
  
 int n = arr.length;  
  
 for (int i = 0; i < n - 1; i++) {  
 for (int j = 0; j < n - i - 1; j++) {  
  
 if (arr[j] > arr[j + 1]) {  
 int temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 }  
 }  
 }  
}Copy

There is also an optimized version of it that relies on a while loop. You can find it in the code bundled to this book under the name bubbleSortOptimized().

As a performance comparison of time execution, for a random array of 100,000 integers, the optimized version will work around 2 seconds faster.

The preceding implementations work well for sorting arrays of primitives, but, for sorting an array of Object, we need to bring Comparator into the code, as follows:

public static <T> void bubbleSortWithComparator(  
 T arr[], Comparator<? super T> c) {  
  
 int n = arr.length;  
  
 for (int i = 0; i < n - 1; i++) {  
 for (int j = 0; j < n - i - 1; j++) {  
  
 if (c.compare(arr[j], arr[j + 1]) > 0) {  
 T temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 }  
 }  
 }  
}Copy

Remember the Melon class from before? Well, we can write a Comparator for it by implementing the Comparator interface:

public class MelonComparator implements Comparator<Melon> {  
  
 @Override  
 public int compare(Melon o1, Melon o2) {  
 return o1.getType().compareTo(o2.getType());  
 }  
}Copy

Or, in Java 8 functional style, we have the following:

// Ascending  
Comparator<Melon> byType = Comparator.comparing(Melon::getType);  
  
// Descending  
Comparator<Melon> byType   
 = Comparator.comparing(Melon::getType).reversed();Copy

Having an array of Melon, the preceding Comparator, and the bubbleSortWithComparator() method in a utility class named ArraySorts, we can write something along the lines of the following:

Melon[] melons = {...};  
ArraySorts.bubbleSortWithComparator(melons, byType);Copy

For brevity, the Bubble sort optimized version with a Comparator was skipped, but it is available in the code bundled to the book.

Bubble sort is fast when the array is almost sorted. Also, it fits well for sorting *rabbits* (big elements that are close to the start of the array) and *turtles* (small elements that are close to the end of the array). But overall, this is a slow algorithm.

Insertion sort

The insertion sort algorithm relies on a simple flow. It starts from the second element and compares it with the element before. If the element before is greater than the current element, then the algorithm swaps the elements. This process continues until the element before is smaller than the current element.

In that case, the algorithm passes to the next element in the array and repeats the flow, as in the following diagram:

Forma, Rectángulo

Descripción generada automáticamente

The time complexity cases are as follows: best case O(n), average case O(n2), worst case O(n2)

The space complexity case is as follows: worst case O(1)

Based on this flow, an implementation for primitive types will be as follows:

public static void insertionSort(int arr[]) {  
  
 int n = arr.length;  
  
 for (int i = 1; i < n; ++i) {  
  
 int key = arr[i];  
 int j = i - 1;  
  
 while (j >= 0 && arr[j] > key) {  
 arr[j + 1] = arr[j];  
 j = j - 1;  
 }  
  
 arr[j + 1] = key;  
 }  
}Copy

For comparing an array of Melon, we need to bring a Comparator in to the implementation as follows:

public static <T> void insertionSortWithComparator(  
 T arr[], Comparator<? super T> c) {  
  
 int n = arr.length;  
  
 for (int i = 1; i < n; ++i) {  
  
 T key = arr[i];  
 int j = i - 1;  
  
 while (j >= 0 && c.compare(arr[j], key) > 0) {  
 arr[j + 1] = arr[j];  
 j = j - 1;  
 }  
  
 arr[j + 1] = key;  
 }  
}Copy

Here, we have a Comparator that sorts the melons by type and weight written in Java 8 functional style using the thenComparing() method:

Comparator<Melon> byType = Comparator.comparing(Melon::getType)  
 .thenComparing(Melon::getWeight);Copy

Having an array of Melon, the preceding Comparator, and the insertionSortWithComparator() method in a utility class named ArraySorts, we can write something as follows:

Melon[] melons = {...};  
ArraySorts.insertionSortWithComparator(melons, byType);Copy

This can be fast for small and mostly sorted arrays. Also, it performs well when adding new elements to an array. It is also very memory-efficient since a single element is moved around.

Counting sort

The counting sort flow starts by calculating the minimum and the maximum element in the array. Based on the computed minimum and maximum, the algorithm defines a new array that will be used to count the unsorted elements by using the *element* as the *index*. Furthermore, this new array is modified in such a way that each *element* at each *index* stores the sum of previous counts. Finally, the sorted array is obtained from this new array.

The time complexity cases are as follows: best case O(n + k), average case O(n + k), worst case O(n + k)

The space complexity case is as follows: worst case O(k)

k is the number of possible values in the range.  
n is the number of elements to be sorted.

Let's consider a quick example. The initial array contains the following elements, arr: **4**, **2**, **6**, **2**, **6**, **8**, **5**:

Diagrama

Descripción generada automáticamente

The minimum element is **2** and the maximum element is **8**. The new array, counts, will have a size equal to the maximum minus the minimum + 1 = 8 - 2 + 1 = 7.

Counting each element will result in the following array (counts[arr[i] - min]++):

counts[2] = 1 (4); counts[0] = 2 (2); counts[4] = 2 (6);  
counts[6] = 1 (8); counts[3] = 1 (5);Copy

Now, we must loop this array and use it to reconstruct the sorted array as in the following implementation:

public static void countingSort(int[] arr) {  
  
 int min = arr[0];  
 int max = arr[0];  
  
 for (int i = 1; i < arr.length; i++) {  
 if (arr[i] < min) {  
 min = arr[i];  
 } else if (arr[i] > max) {  
 max = arr[i];  
 }  
 }  
  
 int[] counts = new int[max - min + 1];  
  
 for (int i = 0; i < arr.length; i++) {  
 counts[arr[i] - min]++;  
 }  
  
 int sortedIndex = 0;  
  
 for (int i = 0; i < counts.length; i++) {  
 while (counts[i] > 0) {  
 arr[sortedIndex++] = i + min;  
 counts[i]--;  
 }  
 }  
}Copy

This is a very fast algorithm.

Heap sort

Heap sort is an algorithm that relies on a binary heap (complete binary tree).

The time complexity cases are as follows: best case O(n log n), average case O(n log n), worst case O(n log n)

The space complexity case is as follows: worst case O(1)

Sorting elements in ascending order can be accomplished via a *Max Heap* (the parent node is always greater than, or equal to, child nodes), and in descending order via a *Min Heap* (the parent node is always smaller than, or equal to, child nodes).

At the first step, the algorithm uses the array provided to build this heap and transform it into a *Max Heap* (the heap is represented by another array). Since this is a *Max Heap*, the largest element is the root of the heap. At the next step, the root is swapped with the last element from the heap and the heap size is reduced by 1 (delete the last node from the heap). The elements that are at the top of the heap come out in sorted order. The final step consists of *heapify* (the recursive process that builds the heap in a top-down manner), and the root of the heap (reconstruct the *Max Heap*). These three steps are repeated until the heap size is greater than 1:

Imagen que contiene Forma

Descripción generada automáticamente

For example, let's assume the array from the preceding diagram – **4**, **5**, **2**, **7**, **1**:

1. So, at the first step, we build the heap: **4**, **5**, **2**, **7**, **1**.
2. We build the *Max Heap*: **7**, **5**, **2**, **4**, **1** (we swapped **5** with **4**, **4** with **7**, and **5** with **7**).
3. Next, we swap the root (**7**) with the last element (**1**) and delete **7**. Result: **1**, **5**, **2**, **4**, **7**.
4. Further, we construct the *Max Heap* again: **5**, **4**, **2**, **1** (we swapped **5** with **1** and **1** with **4**).
5. We swap the root (**5**) with the last element (**1**) and delete **5**. Result: **1**, **4**, **2**, **5**, **7**.
6. Next, we construct the *Max Heap* again: **4**, **1**, **2** (we swapped **1** with **4**).
7. We swap the root (**4**) with the last element (**2**) and delete **4**. Result: **2**, **1**.
8. This is a *Max Heap*, so swap the root (**2**) with the last element (**1**) and remove **2**: **1**, **2**, **4**, **5**, **7**.
9. Done! There is a single element left in the heap (**1**).

In code lines, the preceding example can be generalized as follows:

public static void heapSort(int[] arr) {  
 int n = arr.length;  
  
 buildHeap(arr, n);  
  
 while (n > 1) {  
 swap(arr, 0, n - 1);  
 n--;  
 heapify(arr, n, 0);  
 }  
}  
  
private static void buildHeap(int[] arr, int n) {  
 for (int i = arr.length / 2; i >= 0; i--) {  
 heapify(arr, n, i);  
 }  
}  
  
private static void heapify(int[] arr, int n, int i) {  
 int left = i \* 2 + 1;  
 int right = i \* 2 + 2;  
 int greater;  
  
 if (left < n && arr[left] > arr[i]) {  
 greater = left;  
 } else {  
 greater = i;  
 }  
  
 if (right < n && arr[right] > arr[greater]) {  
 greater = right;  
 }  
  
 if (greater != i) {  
 swap(arr, i, greater);  
 heapify(arr, n, greater);  
 }  
}  
  
private static void swap(int[] arr, int x, int y) {  
 int temp = arr[x];  
 arr[x] = arr[y];  
 arr[y] = temp;  
}Copy

If we want to compare objects, then we have to bring a Comparator into the implementation. This solution is available in the code bundled to this book under the name heapSortWithComparator().

Here, it is a Comparator written in Java 8 functional style that uses the thenComparing() and reversed() methods to sort the melons in descending order by type and weight:

Comparator<Melon> byType = Comparator.comparing(Melon::getType)  
 .thenComparing(Melon::getWeight).reversed(); Copy

Having an array of Melon, the preceding Comparator, and the heapSortWithComparator() method in a utility class named ArraySorts, we can write something as follows:

Melon[] melons = {...};  
ArraySorts.heapSortWithComparator(melons, byType);Copy

Heap sort is pretty fast, but is not stable. For example, sorting an array that is already sorted may leave it in a different order.

We will stop our dissertation regarding sorting arrays here, but, in the code bundled to this book, there are a few more sorting algorithms available:

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente**

There are many other algorithms dedicated to sorting arrays. Some of them are built on top of those presented here (for example, Comb sort, Cocktail sort, and Odd-even sort are flavors of Bubble sort, Bucket sort is a distribution sort commonly relying on Insertion sort, Radix sort (LSD) is a stable distribution similar to Bucket sort, and Gnome sort is a variation of Insertion sort).

Others are different approaches (for example, Quicksort implemented by the Arrays.sort() method, and Merge sort implemented by Arrays.parallelSort()).

By way of a bonus to this section, let's see how we can shuffle an array. An efficient way to accomplish this relies on the Fisher-Yates shuffle (known as the Knuth shuffle). Basically, we loop the array in reverse order and we randomly swap elements. For primitives (for example, int), the implementation is as follows:

public static void shuffleInt(int[] arr) {  
  
 int index;  
  
 Random random = new Random();  
  
 for (int i = arr.length - 1; i > 0; i--) {  
  
 index = random.nextInt(i + 1);  
 swap(arr, index, i);  
 }  
}Copy

In the code bundled to this book, there is also the implementation for shuffling an array of Object.

Shuffling a list is pretty straightforward via Collections.shuffle(List<?> list).

100. Finding an element in an array

When we search for an element in an array, we may be interested to find out the index at which this element occurs, or only whether it is present in the array. The solutions presented in this section are materialized in the methods from the following screenshot:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Let's take a look at the different solutions in the next sections.

Check only for the presence

Let's assume the following array of integers:

int[] numbers = {4, 5, 1, 3, 7, 4, 1};Copy

Since this is an array of primitives, the solution can simply loop the array and return to the first occurrence of the given integer, as follows:

public static boolean containsElement(int[] arr, int toContain) {  
  
 for (int elem: arr) {  
 if (elem == toContain) {  
 return true;  
 }  
 }  
  
 return false;  
}Copy

Another solution to this problem can rely on the Arrays.binarySearch() methods. There are several flavors of this method, but in this case, we need this one: int binarySearch​(int[] a, int key). The method will search the given key in the given array and will return the corresponding index or a negative value. The only issue is that this method works only for sorted arrays; therefore, we need to sort the array beforehand:

public static boolean containsElement(int[] arr, int toContain) {  
  
 Arrays.sort(arr);  
 int index = Arrays.binarySearch(arr, toContain);  
  
 return (index >= 0);  
}Copy

If the array is already sorted, then the preceding method can be optimized by removing the sorting step. Moreover, if the array is sorted, the preceding method may return the index where the element occurs in the array instead of a boolean. However, if the array is not sorted, then keep in mind that the returned index corresponds to the sorted array, not to the unsorted (initial) array. If you don't want to sort the initial array, then it is advisable to pass a clone of the array to this method. Another approach will be to clone the array inside this helper method.

In Java 8, the solution can rely on a functional style approach. A good candidate here is the anyMatch() method. This method returns whether any elements of the stream match the predicate provided. So, all we need to do is to convert the array to a stream, as follows:

public static boolean containsElement(int[] arr, int toContain) {  
  
 return Arrays.stream(arr)  
 .anyMatch(e -> e == toContain);  
}Copy

For any other primitive type, it is pretty straightforward to adapt or generalize the preceding examples.

Now, let's focus on finding Object in arrays. Let's consider the Melon class:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals() and hashCode() skipped for brevity  
}Copy

Next, let's consider an array of Melon:

Melon[] melons = new Melon[] {new Melon("Crenshaw", 2000),  
 new Melon("Gac", 1200), new Melon("Bitter", 2200)  
};Copy

Now, let's assume that we want to find the Gac melon of 1,200 g in this array. A solution can rely on the equals() method, which is used to determine the equality of two objects:

public static <T> boolean   
 containsElementObject(T[] arr, T toContain) {  
  
 for (T elem: arr) {  
 if (elem.equals(toContain)) {  
 return true;  
 }  
 }  
  
 return false;  
}Copy

Similarly, we can rely on Arrays.asList(arr).contains(find). Basically, convert the array to a List and call the contains() method. Behind the scenes, this method uses the equals() contract.

If this method lives in a utility class named ArraySearch, then the following call will return true:

// true  
boolean found = ArraySearch.containsElementObject(  
 melons, new Melon("Gac", 1200));Copy

This solution works fine as long as we want to rely on the equals() contract. But we may consider that our melon is present in the array if its name occurs (Gac), or if its weight occurs (1,200). For such cases, it is more practical to rely on a Comparator:

public static <T> boolean containsElementObject(  
 T[] arr, T toContain, Comparator<? super T> c) {  
  
 for (T elem: arr) {  
 if (c.compare(elem, toContain) == 0) {  
 return true;  
 }  
 }  
  
 return false;  
}Copy

Now, a Comparator that takes into account only the type of melons can be written as follows:

Comparator<Melon> byType = Comparator.comparing(Melon::getType);Copy

Since the Comparator ignores the weight of the melon (there is no melon of 1,205 grams), the following invocation will return true:

// true  
boolean found = ArraySearch.containsElementObject(  
 melons, new Melon("Gac", 1205), byType);Copy

Another approach relies on another flavor of binarySearch(). The Arrays class provides a binarySearch() method that gets a Comparator, <T> int binarySearch(T[] a, T key, Comparator<? super T> c). This means that we can use it as follows:

public static <T> boolean containsElementObject(  
 T[] arr, T toContain, Comparator<? super T> c) {  
  
 Arrays.sort(arr, c);  
 int index = Arrays.binarySearch(arr, toContain, c);  
  
 return (index >= 0);  
}Copy

If the initial array state should remain unmodified, then it is advisable to pass a clone of the array to this method. Another approach would be to clone the array inside this helper method.

Now, a Comparator that takes into account only the weight of melons can be written as follows:

Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);Copy

Since the Comparator ignores the type of melon (there is no melon of the Honeydew type), the following invocation will return true:

// true  
boolean found = ArraySearch.containsElementObject(  
 melons, new Melon("Honeydew", 1200), byWeight);Copy

Check only for the first index

For an array of primitives, the simplest implementation speaks for itself:

public static int findIndexOfElement(int[] arr, int toFind) {  
  
 for (int i = 0; i < arr.length; i++) {  
 if (arr[i] == toFind) {  
 return i;  
 }  
 }  
  
 return -1;  
}Copy

Relying on Java 8 functional style, we can try to loop the array and filter the elements that match the given element. In the end, simply return the first found element:

public static int findIndexOfElement(int[] arr, int toFind) {  
  
 return IntStream.range(0, arr.length)  
 .filter(i -> toFind == arr[i])  
 .findFirst()  
 .orElse(-1);  
}Copy

For an array of Object, there are at least three approaches. In the first instance, we can rely on the equals() contract:

public static <T> int findIndexOfElementObject(T[] arr, T toFind) {  
  
 for (int i = 0; i < arr.length; i++) {  
 if (arr[i].equals(toFind)) {  
 return i;  
 }  
 }  
  
 return -1;  
}Copy

Similarly, we can rely on Arrays.asList(arr).indexOf(find). Basically, convert the array to a List and call the indexOf() method. Behind the scenes, this method uses the equals() contract.

Secondly, we can rely on a Comparator:

public static <T> int findIndexOfElementObject(  
 T[] arr, T toFind, Comparator<? super T> c) {  
  
 for (int i = 0; i < arr.length; i++) {  
 if (c.compare(arr[i], toFind) == 0) {  
 return i;  
 }  
 }  
  
 return -1;  
}Copy

And thirdly, we can rely on Java 8 functional style and a Comparator:

public static <T> int findIndexOfElementObject(  
 T[] arr, T toFind, Comparator<? super T> c) {  
  
 return IntStream.range(0, arr.length)  
 .filter(i -> c.compare(toFind, arr[i]) == 0)  
 .findFirst()  
 .orElse(-1);  
}Copy

101. Checking whether two arrays are equal or mismatches

Two arrays of primitives are equal if they contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal.

The solutions to these two problems rely on the Arrays utility class. The following sections give the solutions to these problems.

Checking whether two arrays are equal

Checking whether two arrays are equal can be easily accomplished via the Arrays.equals() method. This flag method comes in many flavors for primitive types, Object, and generics. It also supports comparators.

Let's consider the following three arrays of integers:

int[] integers1 = {3, 4, 5, 6, 1, 5};  
int[] integers2 = {3, 4, 5, 6, 1, 5};  
int[] integers3 = {3, 4, 5, 6, 1, 3};Copy

Now, let's check whether integers1 is equal to integers2, and whether integers1 is equal to integers3. This is very simple:

boolean i12 = Arrays.equals(integers1, integers2); // true  
boolean i13 = Arrays.equals(integers1, integers3); // falseCopy

The preceding examples check whether two arrays are equal, but we can check whether two segments (or ranges) of the arrays are equal as well via the boolean equals(int[] a, int aFromIndex, int aToIndex, int[] b, int bFromIndex, int bToIndex) method. So, we demarcate the segment of the first array via the range [aFromIndex, aToIndex) and the segment of the second array via the range [bFromIndex, bToIndex):

// true  
boolean is13 = Arrays.equals(integers1, 1, 4, integers3, 1, 4);Copy

Now, let's assume three arrays of Melon:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 public Melon(String type, int weight) {  
 this.type = type;  
 this.weight = weight;  
 }  
  
 // getters, equals() and hashCode() omitted for brevity  
}  
  
Melon[] melons1 = {  
 new Melon("Horned", 1500), new Melon("Gac", 1000)  
};  
  
Melon[] melons2 = {  
 new Melon("Horned", 1500), new Melon("Gac", 1000)  
};  
  
Melon[] melons3 = {  
 new Melon("Hami", 1500), new Melon("Gac", 1000)  
};Copy

Two arrays of Object are considered equal based on the equals() contract, or based on the specified Comparator. We can easily check whether melons1 is equal to melons2, and whether melons1 is equal to melons3 as follows:

boolean m12 = Arrays.equals(melons1, melons2); // true  
boolean m13 = Arrays.equals(melons1, melons3); // falseCopy

And, in an explicit range, use boolean equals(Object[] a, int aFromIndex, int aToIndex, Object[] b, int bFromIndex, int bToIndex):

boolean ms13 = Arrays.equals(melons1, 1, 2, melons3, 1, 2); // falseCopy

While these examples rely on the Melon.equals() implementation, the following two examples rely on the following two Comparator:

Comparator<Melon> byType = Comparator.comparing(Melon::getType);  
Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);Copy

Using the boolean equals(T[] a, T[] a2, Comparator<? super T> cmp), we have the following:

boolean mw13 = Arrays.equals(melons1, melons3, byWeight); // true  
boolean mt13 = Arrays.equals(melons1, melons3, byType); // falseCopy

And, in an explicit range, using Comparator, <T> boolean equals(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex, Comparator<? super T> cmp), we have the following:

// true  
boolean mrt13 = Arrays.equals(melons1, 1, 2, melons3, 1, 2, byType);Copy

Checking whether two arrays contain a mismatch

If two arrays are equal, then a mismatch should return -1. But if two arrays are not equal, then a mismatch should return the index of the first mismatch between the two given arrays. In order to resolve this problem, we can rely on JDK 9 Arrays.mismatch() methods.

For example, we can check for mismatches between integers1 and integers2 as follows:

int mi12 = Arrays.mismatch(integers1, integers2); // -1Copy

The result is -1, since integers1 and integers2 are equal. But if we check for integers1 and integers3, we receive the value 5, which is the index of the first mismatch between these two:

int mi13 = Arrays.mismatch(integers1, integers3); // 5Copy

If the given arrays have different lengths and the smaller one is a prefix for the larger one, then the returned mismatch is the length of the smaller array.

For arrays of Object, there are dedicated mismatch() methods as well. These methods count on the equals() contract or on the given Comparator. We can check whether there is a mismatch between melons1 and melons2 as follows:

int mm12 = Arrays.mismatch(melons1, melons2); // -1Copy

If the mismatch occurs on the first index, then the returned value is 0. This is happening in the case of melons1 and melons3:

int mm13 = Arrays.mismatch(melons1, melons3); // 0Copy

As in the case of Arrays.equals(), we can check mismatches in an explicit range using a Comparator:

// range [1, 2), return -1  
int mms13 = Arrays.mismatch(melons1, 1, 2, melons3, 1, 2);  
  
// Comparator by melon's weights, return -1  
int mmw13 = Arrays.mismatch(melons1, melons3, byWeight);  
  
// Comparator by melon's types, return 0  
int mmt13 = Arrays.mismatch(melons1, melons3, byType);  
  
// range [1,2) and Comparator by melon's types, return -1  
int mmrt13 = Arrays.mismatch(melons1, 1, 2, melons3, 1, 2, byType);Copy

102. Comparing two arrays lexicographically

Starting with JDK 9, we can compare two arrays lexicographically via the Arrays.compare() methods. Since there is no need to reinvent the wheel, just upgrade to JDK 9 and let's dive into it.

A lexicographic comparison of two arrays may return the following:

* 0, if the given arrays are equal and contain the same elements in the same order
* A value less than 0 if the first array is lexicographically less than the second array
* A value greater than 0 if the first array is lexicographically greater than the second array

If the first array length is less than the second array length, then the first array is lexicographically less than the second array. If the arrays have the same length, contain primitives, and share a common prefix, then the lexicographic comparison is the result of comparing two elements, precisely as Integer.compare(int, int), Boolean.compare(boolean, boolean), Byte.compare(byte, byte), and so on. If the arrays contain Object, then the lexicographic comparison is relying on the given Comparator or on the Comparable implementation.

First, let's consider the following arrays of primitives:

int[] integers1 = {3, 4, 5, 6, 1, 5};  
int[] integers2 = {3, 4, 5, 6, 1, 5};  
int[] integers3 = {3, 4, 5, 6, 1, 3};Copy

Now, integers1 is lexicographically equal to integers2 because they are equal and contain the same elements in the same order, int compare(int[] a, int[] b):

int i12 = Arrays.compare(integers1, integers2); // 0Copy

However, integers1 is lexicographically greater than integers3, since they share the same prefix (3, 4, 5, 6, 1), but for the last element, Integer.compare(5,3) returns a value greater than 0 since 5 is greater than 3:

int i13 = Arrays.compare(integers1, integers3); // 1Copy

A lexicographical comparison can be accomplished on different ranges of the arrays. For example, the following example compares integers1 and integers3 in the range [3, 6) via the int compare(int[] a, int aFromIndex, int aToIndex, int[] b, int bFromIndex, int bToIndex) method:

int is13 = Arrays.compare(integers1, 3, 6, integers3, 3, 6); // 1Copy

For arrays of Object, the Arrays class also provides a set of dedicated compare() methods. Remember the Melon class? Well, in order to compare two arrays of Melon without an explicit Comparator, we need to implement the Comparable interface and implement the compareTo() method. Let's assume that we are relying on melon weights as follows:

public class Melon implements Comparable {  
  
 private final String type;  
 private final int weight;  
  
 @Override  
 public int compareTo(Object o) {  
 Melon m = (Melon) o;  
  
 return Integer.compare(this.getWeight(), m.getWeight());  
 }  
  
 // constructor, getters, equals() and hashCode() omitted for brevity  
}Copy

Note that the lexicographic comparison of arrays of Object doesn't rely on equals(). It requires an explicit Comparator or Comparable elements.

Let's assume the following arrays of Melon:

Melon[] melons1 = {new Melon("Horned", 1500), new Melon("Gac", 1000)};  
Melon[] melons2 = {new Melon("Horned", 1500), new Melon("Gac", 1000)};  
Melon[] melons3 = {new Melon("Hami", 1600), new Melon("Gac", 800)};Copy

And, let's compare lexicographically melons1 with melons2 via <T extends Comparable<? super T>> int compare(T[] a, T[] b):

int m12 = Arrays.compare(melons1, melons2); // 0Copy

Since melons1 and melons2 are identical, the result is 0.

Now, let's do the same thing with melons1 and melons3. This time, the result will be negative, which means that melons1 is lexicographically less than melons3. This is true since, at index 0, the Horned melon has a weight of 1,500 g, which is less than the weight of the Hami melon, which is 1,600 g:

int m13 = Arrays.compare(melons1, melons3); // -1Copy

We can perform the comparison in different ranges of the arrays via the <T extends Comparable<? super T>> int compare(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex) method. For example, in the common range [1, 2), melons1 is lexicographically greater than melons2, since the weight of Gac is 1,000g in melons1 and 800g in melons3:

int ms13 = Arrays.compare(melons1, 1, 2, melons3, 1, 2); // 1Copy

If we don't want to rely on Comparable elements (implement Comparable), we can pass in a Comparator via the <T> int compare(T[] a, T[] b, Comparator<? super T> cmp) method:

Comparator<Melon> byType = Comparator.comparing(Melon::getType);  
int mt13 = Arrays.compare(melons1, melons3, byType); // 14Copy

Using ranges is also possible by means of <T> int compare(T[] a, int aFromIndex, int aToIndex, T[] b, int bFromIndex, int bToIndex, Comparator<? super T> cmp):

int mrt13 = Arrays.compare(melons1, 1, 2, melons3, 1, 2, byType); // 0Copy

If the arrays of numbers should be treated unsigned, then rely on the bunch of Arrays.compareUnsigned​() methods, which are available for byte, short, int, and long.  
  
To compare two strings lexicographically, rely on String.compareTo() and int compareTo(String anotherString).

103. Creating a Stream from an array

Once we create a Stream from an array, we have access to all the Stream API goodies. Therefore, this is a handy operation that is important to have in our tool belt.

Let's start with an array of strings (can be other objects as well):

String[] arr = {"One", "Two", "Three", "Four", "Five"};Copy

The easiest way to create Stream from this String[] array is to rely on the Arrays.stream() method available starting with JDK 8:

Stream<String> stream = Arrays.stream(arr);Copy

Or, if we need a stream from a sub-array, then simply add the range as arguments. For example, let's create a Stream from the elements that range between (0,2), which are one and two:

Stream<String> stream = Arrays.stream(arr, 0, 2);Copy

The same cases, but passing through a List, can be written as follows:

Stream<String> stream = Arrays.asList(arr).stream();  
Stream<String> stream = Arrays.asList(arr).subList(0, 2).stream();Copy

Another solution relies on Stream.of() methods, as in the following straightforward examples:

Stream<String> stream = Stream.of(arr);  
Stream<String> stream = Stream.of("One", "Two", "Three");Copy

Creating an array from a Stream can be accomplished via the Stream.toArray() method. For example, a simple approach appears as follows:

String[] array = stream.toArray(String[]::new);Copy

In addition, let's consider an array of primitives:

int[] integers = {2, 3, 4, 1};Copy

In such a case, the Arrays.stream() method can help again, the only difference being that the returned result is of the IntStream type (this is the int primitive specialization of Stream):

IntStream intStream = Arrays.stream(integers);Copy

But the IntStream class also provides an of() method that can be used as follows:

IntStream intStream = IntStream.of(integers);Copy

Sometimes, we need to define a Stream of sequentially ordered integers with an incremental step of 1. Moreover, the size of the Stream should be equal to the size of an array. Especially for such cases, the IntStream method provides two methods—range(int inclusive, int exclusive) and rangeClosed(int startInclusive, int endInclusive):

IntStream intStream = IntStream.range(0, integers.length);  
IntStream intStream = IntStream.rangeClosed(0, integers.length);Copy

Creating an array from a Stream of integers can be accomplished via the Stream.toArray() method. For example, a simple approach appears as follows:

int[] intArray = intStream.toArray();  
  
// for boxed integers  
int[] intArray = intStream.mapToInt(i -> i).toArray();Copy

Besides the IntStream specialization of Stream, JDK 8 provides specializations for long (LongStream) and double (DoubleStream).

104. Minimum, maximum, and average of an array

Computing the minimum, maximum, and average values of an array is a common task. Let's look at several approaches to solving this problem in functional style and imperative programming.

Computing maximum and minimum

Computing the maximum value of an array of numbers can be implemented by looping the array and tracking the maximum value via a comparison with each element of the array. In terms of lines of code, this can be written as follows:

public static int max(int[] arr) {  
  
 int max = arr[0];  
  
 for (int elem: arr) {  
 if (elem > max) {  
 max = elem;  
 }  
 }  
  
 return max;  
}Copy

A little pinch in readability here may entail using the Math.max() method instead of an if statement:

...  
max = Math.max(max, elem);  
...Copy

Let's suppose that we have the following array of integers and a utility class named MathArrays that contains the preceding methods:

int[] integers = {2, 3, 4, 1, -4, 6, 2};Copy

The maximum of this array can easily be obtained as follows:

int maxInt = MathArrays.max(integers); // 6Copy

In Java 8 functional style, the solution to this problem entails a single line of code:

int maxInt = Arrays.stream(integers).max().getAsInt();Copy

In the functional-style approach, the max() method returns an OptionalInt. Similarly, we have OptionalLong and OptionalDouble.

Furthermore, let's assume an array of objects, in this case, an array of Melon:

Melon[] melons = {  
 new Melon("Horned", 1500), new Melon("Gac", 2200),  
 new Melon("Hami", 1600), new Melon("Gac", 2100)  
};  
  
public class Melon implements Comparable {  
  
 private final String type;  
 private final int weight;  
  
 @Override  
 public int compareTo(Object o) {  
 Melon m = (Melon) o;  
  
 return Integer.compare(this.getWeight(), m.getWeight());  
 }  
  
 // constructor, getters, equals() and hashCode() omitted for brevity  
}Copy

It is obvious that our max() methods defined earlier cannot be used in this case, but the logical principle remains the same. This time, the implementation should rely on Comparable or Comparator. The implementation based on Comparable can be as follows:

public static <T extends Comparable<T>> T max(T[] arr) {  
  
 T max = arr[0];  
  
 for (T elem : arr) {  
 if (elem.compareTo(max) > 0) {  
 max = elem;  
 }  
 }  
  
 return max;  
}Copy

Check the Melon.compareTo() method and note that our implementation will compare the weights of melons. Therefore, we can easily find the heaviest melon from our array as follows:

Melon maxMelon = MathArrays.max(melons); // Gac(2200g)Copy

And the implementation relying on Comparator can be written as follows:

public static <T> T max(T[] arr, Comparator<? super T> c) {  
  
 T max = arr[0];  
  
 for (T elem: arr) {  
 if (c.compare(elem, max) > 0) {  
 max = elem;  
 }  
 }  
  
 return max;  
}Copy

And, if we define a Comparator according to the type of melon, we have the following:

Comparator<Melon> byType = Comparator.comparing(Melon::getType);Copy

Then, we get the maximum melon conforming to the lexicographical comparison of strings:

Melon maxMelon = MathArrays.max(melons, byType); // Horned(1500g)Copy

In Java 8 functional style, the solution to this problem entails a single line of code:

Melon maxMelon = Arrays.stream(melons).max(byType).orElseThrow();Copy

Computing average

Computing the average value of an array of numbers (in this case integers) can be implemented in two simple steps:

1. Compute the sum of the elements from the array.
2. Divide this sum by the length of the array.

In code lines, we have the following:

public static double average(int[] arr) {  
  
 return sum(arr) / arr.length;  
}  
  
public static double sum(int[] arr) {  
  
 double sum = 0;  
  
 for (int elem: arr) {  
 sum += elem;  
 }  
  
 return sum;  
}Copy

The average of our integers array is 2.0:

double avg = MathArrays.average(integers);Copy

In Java 8 functional style, the solution to this problem entails a single line of code:

double avg = Arrays.stream(integers).average().getAsDouble();Copy

For third-party library support, please consider Apache Common Lang (ArrayUtil) and Guava's Chars, Ints, Longs, and other classes.

105. Reversing an array

There are several solutions to this problem. Some of them mutate the initial array, while others just return a new array.

Let's assume the following array of integers:

int[] integers = {-1, 2, 3, 1, 4, 5, 3, 2, 22};Copy

Let's start with a simple implementation that swaps the first element of the array with the last element, the second element with the penultimate element, and so on:

public static void reverse(int[] arr) {  
  
 for (int leftHead = 0, rightHead = arr.length - 1;   
 leftHead < rightHead; leftHead++, rightHead--) {  
  
 int elem = arr[leftHead];  
 arr[leftHead] = arr[rightHead];  
 arr[rightHead] = elem;  
 }  
}Copy

The preceding solution mutates the given array and this is not always the desired behavior. Of course, we can modify it to return a new array, or we can rely on Java 8 functional style as follows:

// 22, 2, 3, 5, 4, 1, 3, 2, -1  
int[] reversed = IntStream.rangeClosed(1, integers.length)  
 .map(i -> integers[integers.length - i]).toArray();Copy

Now, let's reverse an array of objects. For this, let's consider the Melon class:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals(), hashCode() omitted for brevity  
}Copy

Also, let's consider an array of Melon:

Melon[] melons = {  
 new Melon("Crenshaw", 2000),   
 new Melon("Gac", 1200),  
 new Melon("Bitter", 2200)  
};Copy

The first solution entails using generics to shape the implementation that swaps the first element of the array with the last element, the second element with the second last element, and so on:

public static <T> void reverse(T[] arr) {  
  
 for (int leftHead = 0, rightHead = arr.length - 1;   
 leftHead < rightHead; leftHead++, rightHead--) {  
  
 T elem = arr[leftHead];  
 arr[leftHead] = arr[rightHead];  
 arr[rightHead] = elem;  
 }  
}Copy

Since our array contains objects, we can rely on Collections.reverse() as well. We just need to convert the array to a List via the Arrays.asList() method:

// Bitter(2200g), Gac(1200g), Crenshaw(2000g)  
Collections.reverse(Arrays.asList(melons));Copy

The preceding two solutions mutate the elements of the array. Java 8 functional style can help us to avoid this mutation:

// Bitter(2200g), Gac(1200g), Crenshaw(2000g)  
Melon[] reversed = IntStream.rangeClosed(1, melons.length)  
 .mapToObj(i -> melons[melons.length - i])  
 .toArray(Melon[]:new);Copy

For third-party library support, please consider Apache Common Lang (ArrayUtils.reverse()) and Guava's Lists class.

106. Filling and setting an array

Sometimes, we need to fill up an array with a fixed value. For example, we may want to fill up an array of integers with the value 1. The simplest way to accomplish this relies on a for statement as follows:

int[] arr = new int[10];  
  
// 1, 1, 1, 1, 1, 1, 1, 1, 1, 1  
for (int i = 0; i < arr.length; i++) {  
 arr[i] = 1;  
}Copy

But we can reduce this code to a single line of code by means of the Arrays.fill() methods. This method comes in different flavors for primitives and for objects. The preceding code can be rewritten via Arrays.fill(int[] a, int val) as follows:

// 1, 1, 1, 1, 1, 1, 1, 1, 1, 1  
Arrays.fill(arr, 1);Copy

Arrays.fill() also come with flavors for filling up just a segment/range of an array. For integers, this method is fill​(int[] a, int fromIndexInclusive, int toIndexExclusive, int val).

Now, how about applying a generator function to compute each element of the array? For example, let's assume that we want to compute each element as the previous one plus 1. The simplest approach will again rely on a for statement as follows:

// 1, 2, 3, 4, 5, 6, 7, 8, 9, 10  
for (int i = 1; i < arr.length; i++) {  
 arr[i] = arr[i - 1] + 1;  
}Copy

The preceding code has to be modified accordingly depending on the computations that need to be applied to each element.

For such tasks, JDK 8 comes with a bunch of Arrays.setAll() and Arrays.parallelSetAll() methods. For example, the preceding snippet of code can be rewritten via setAll​(int[] array, IntUnaryOperator generator) as follows:

// 1, 2, 3, 4, 5, 6, 7, 8, 9, 10  
Arrays.setAll(arr, t -> {  
 if (t == 0) {  
 return arr[t];  
 } else {  
 return arr[t - 1] + 1;  
 }  
});Copy

Besides this method, we also have setAll​(double[] array, IntToDoubleFunction generator), setAll​(long[] array, IntToLongFunction generator), and setAll​(T[] array, IntFunction<? extends T> generator).

Depending on the generator function, this task can be accomplished in parallel or not. For example, the preceding generator function cannot be applied in parallel since each element depends on the value of the preceding element. Trying to apply this generator function in parallel will lead to incorrect and unstable results.

But let's assume that we want to take the preceding array (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) and multiply each even value by itself and decrease each odd value by 1. Since each element can be computed individually, we can empower a parallel process in this case. This is the perfect job for Arrays.parallelSetAll() methods. Basically, these methods are meant to parallelize Arrays.setAll() methods.

Let's now apply parallelSetAll​(int[] array, IntUnaryOperator generator) to this array:

// 0, 4, 2, 16, 4, 36, 6, 64, 8, 100  
Arrays.parallelSetAll(arr, t -> {  
 if (arr[t] % 2 == 0) {  
 return arr[t] \* arr[t];  
 } else {  
 return arr[t] - 1;  
 }  
});Copy

For each Arrays.setAll() method, there is an Arrays.parallelSetAll() method.

As a bonus, Arrays come with a set of methods named parallelPrefix(). These methods are useful for applying a mathematical function to the elements of the array, both cumulatively and concurrently.

For example, if we want to compute each element of the array as the sum of the preceding elements, then we can do it as follows:

// 0, 4, 6, 22, 26, 62, 68, 132, 140, 240  
Arrays.parallelPrefix(arr, (t, q) -> t + q);Copy

107. Next Greater Element

NGE is a classic problem that involves arrays.

Basically, having an array and an element from it, e, we want to fetch the next (right-hand side) element greater than e. For example, let's assume the following array:

int[] integers = {1, 2, 3, 4, 12, 2, 1, 4};Copy

Fetching the NGE for each element will result in the following pairs (-1 is interpreted as no element from the right-hand side is greater than the current one):

1 : 2 2 : 3 3 : 4 4 : 12 12 : -1 2 : 4 1 : 4 4 : -1Copy

A simple solution to this problem will be looping the array for each element until a greater element is found or there are no more elements to check. If we just want to print the pairs on the screen, then we can write a trivial code such as the following:

public static void println(int[] arr) {  
  
 int nge;  
 int n = arr.length;  
  
 for (int i = 0; i < n; i++) {  
 nge = -1;  
 for (int j = i + 1; j < n; j++) {  
 if (arr[i] < arr[j]) {  
 nge = arr[j];  
 break;  
 }  
 }  
  
 System.out.println(arr[i] + " : " + nge);  
 }  
}Copy

Another solution relies on a stack. Mainly, we push elements in the stack until the currently processed element is greater than the top element in the stack. When this is happening, we pop that element. The solution is available in the code bundled to this book.

108. Changing array size

Increasing the size of an array is not straightforward. This is because Java arrays are of a fixed size and we cannot modify their size. The solution to this problem entails creating a new array of the requisite size and copying all the values from the original array to this one. This can be done via the Arrays.copyOf() method or via System.arraycopy() (used internally by Arrays.copyOf()).

For an array of primitives (for example, int), we can add the value to an array after increasing its size by 1 as follows:

public static int[] add(int[] arr, int item) {  
  
 int[] newArr = Arrays.copyOf(arr, arr.length + 1);  
 newArr[newArr.length - 1] = item;  
  
 return newArr;  
}Copy

Or, we can remove the last value as follows:

public static int[] remove(int[] arr) {  
  
 int[] newArr = Arrays.copyOf(arr, arr.length - 1);  
  
 return newArr;  
}Copy

Alternatively, we can resize the array with the given length as follows:

public static int[] resize(int[] arr, int length) {  
  
 int[] newArr = Arrays.copyOf(arr, arr.length + length);  
  
 return newArr;  
}Copy

The code bundled to this book also contains the System.arraycopy() alternatives. Moreover, it contains the implementations for generic arrays. The signatures are as follows:

public static <T> T[] addObject(T[] arr, T item);  
public static <T> T[] removeObject(T[] arr);  
public static <T> T[] resize(T[] arr, int length);Copy

Being in a favorable context, let's bring a related topic into the discussion: how to create a generic array in Java. The following will not work:

T[] arr = new T[*arr\_size*]; // causes generic array creation errorCopy

There are several approaches, but Java uses the following code in copyOf(T[] original, int newLength):

// newType is original.getClass()  
T[] copy = ((Object) newType == (Object) Object[].class) ?  
 (T[]) new Object[newLength] :  
 (T[]) Array.newInstance(newType.getComponentType(), newLength);Copy

109. Creating unmodifiable/immutable collections

Creating unmodifiable/immutable collections in Java can easily be accomplished by means of the Collections.unmodifiable*Foo*() method (for example, unmodifiableList()) and, starting with JDK 9, via the set of of() methods from List, Set, Map, and other interfaces.

Furthermore, we will use these methods in a bunch of examples to obtain unmodifiable/immutable collections. The main goal is to determine whether each defined collection is unmodifiable or immutable.

Before reading this section, it is advisable to read the problems dedicated to immutability from [Chapter 2](https://subscription.packtpub.com/book/programming/9781789801415/5/ch05lvl1sec20/16027e66-3bab-4bff-b106-7cbe15480be6.xhtml), *Objects, Immutability, and Switch Expressions*.

OK. In the case of primitives, it is pretty simple. For example, we can create an immutable List of integers as follows:

private static final List<Integer> LIST   
 = Collections.unmodifiableList(Arrays.asList(1, 2, 3, 4, 5));  
  
private static final List<Integer> LIST = List.of(1, 2, 3, 4, 5);Copy

For the next examples, let's consider the following mutable class:

public class MutableMelon {  
  
 private String type;  
 private int weight;  
  
 // constructor omitted for brevity  
  
 public void setType(String type) {  
 this.type = type;  
 }  
  
 public void setWeight(int weight) {  
 this.weight = weight;  
 }  
  
 // getters, equals() and hashCode() omitted for brevity  
}Copy

Problem 1 (Collections.unmodifiableList())

Let's create a list of MutableMelon via the Collections.unmodifiableList() method:

// Crenshaw(2000g), Gac(1200g)  
private final MutableMelon melon1   
 = new MutableMelon("Crenshaw", 2000);  
private final MutableMelon melon2   
 = new MutableMelon("Gac", 1200);  
  
private final List<MutableMelon> list   
 = Collections.unmodifiableList(Arrays.asList(melon1, melon2));Copy

So, is list unmodifiable or immutable? The answer is unmodifiable. While mutator methods will throw UnsupportedOperationException, the underlying melon1 and melon2 are mutable. For example, let's set the weights of our melons to 0:

melon1.setWeight(0);  
melon2.setWeight(0);Copy

Now, the list will reveal the following melons (so the list was mutated):

Crenshaw(0g), Gac(0g)Copy

Problem 2 (Arrays.asList())

Let's create a list of MutableMelon by hardcoding the instances directly in Arrays.asList():

private final List<MutableMelon> list   
 = Collections.unmodifiableList(Arrays.asList(  
 new MutableMelon("Crenshaw", 2000),   
 new MutableMelon("Gac", 1200)));Copy

So, is the list unmodifiable or immutable? The answer is unmodifiable. While mutator methods will throw UnsupportedOperationException, the hardcoded instances can be accessed via the List.get() method. Once they can be accessed, they can be mutated:

MutableMelon melon1 = list.get(0);  
MutableMelon melon2 = list.get(1);  
  
melon1.setWeight(0);  
melon2.setWeight(0);Copy

Now, the list will reveal the following melons (so the list was mutated):

Crenshaw(0g), Gac(0g)Copy

Problem 3 (Collections.unmodifiableList() and static block)

Let's create a list of MutableMelon via the Collections.unmodifiableList() method and a static block:

private static final List<MutableMelon> list;  
static {  
 final MutableMelon melon1 = new MutableMelon("Crenshaw", 2000);  
 final MutableMelon melon2 = new MutableMelon("Gac", 1200);  
  
 list = Collections.unmodifiableList(Arrays.asList(melon1, melon2));  
}Copy

So, is the list unmodifiable or immutable? The answer is unmodifiable. While mutator methods will throw UnsupportedOperationException, the hardcoded instances can still be accessed via the List.get() method. Once they can be accessed, they can be mutated:

MutableMelon melon1l = list.get(0);  
MutableMelon melon2l = list.get(1);  
  
melon1l.setWeight(0);  
melon2l.setWeight(0);Copy

Now, the list will reveal the following melons (so the list was mutated):

Crenshaw(0g), Gac(0g)Copy

Problem 4 (List.of())

Let's create a list of MutableMelon via List.of():

private final MutableMelon melon1   
 = new MutableMelon("Crenshaw", 2000);  
private final MutableMelon melon2   
 = new MutableMelon("Gac", 1200);  
  
private final List<MutableMelon> list = List.of(melon1, melon2);Copy

So, is the list unmodifiable or immutable? The answer is unmodifiable. While mutator methods will throw UnsupportedOperationException, the hardcoded instances can still be accessed via the List.get() method. Once they can be accessed, they can be mutated:

MutableMelon melon1l = list.get(0);  
MutableMelon melon2l = list.get(1);  
  
melon1l.setWeight(0);  
melon2l.setWeight(0);Copy

Now, the list will reveal the following melons (so the list was mutated):

Crenshaw(0g), Gac(0g)Copy

For the next examples, let's consider the following immutable class:

public final class ImmutableMelon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals() and hashCode() omitted for brevity  
}Copy

Problem 5 (immutable)

Let's now create a list of ImmutableMelon via Collections.unmodifiableList() and the List.of() methods:

private static final ImmutableMelon MELON\_1   
 = new ImmutableMelon("Crenshaw", 2000);  
private static final ImmutableMelon MELON\_2   
 = new ImmutableMelon("Gac", 1200);  
  
private static final List<ImmutableMelon> LIST   
 = Collections.unmodifiableList(Arrays.asList(MELON\_1, MELON\_2));  
private static final List<ImmutableMelon> LIST   
 = List.of(MELON\_1, MELON\_2);Copy

So, is the list unmodifiable or immutable? The answer is immutable. Mutator methods will throw UnsupportedOperationException, and we cannot mutate the instances of ImmutableMelon.

As a rule of thumb, a collection is unmodifiable if it is defined via unmodifiable*Foo*() or of() methods and contains mutable data, and it is immutable if it is unmodifiable and contains immutable data (including primitives).  
  
Pay attention to the fact that impenetrable immutability should take into consideration Java Reflection API and similar APIs that have supplementary powers in manipulating code.  
  
For third-party library support, please consider Apache Common Collection, UnmodifiableList (and companions), and Guava's ImmutableList (and companions).

In the case of Map, we can create an unmodifiable/immutable Map via unmodifiableMap() or the Map.of() methods.

But we can also create an immutable empty Map via Collections.emptyMap():

Map<Integer, MutableMelon> emptyMap = Collections.emptyMap();Copy

Similar to emptyMap(), we have Collections.emptyList(), and Collections.emptySet(). These methods are very handy as returns in methods that return a Map, List, or Set, and we want to avoid returning null.

Alternatively, we can create an unmodifiable/immutable Map with a single element via Collections.singletonMap(K key, V value):

// unmodifiable  
Map<Integer, MutableMelon> mapOfSingleMelon   
 = Collections.singletonMap(1, new MutableMelon("Gac", 1200));  
  
// immutable  
Map<Integer, ImmutableMelon> mapOfSingleMelon   
 = Collections.singletonMap(1, new ImmutableMelon("Gac", 1200));Copy

Similar to singletonMap(), we have singletonList() and singleton(). The latter is for Set.

Moreover, starting with JDK 9, we can create an unmodifiable Map via a method named ofEntries(). This method takes Map.Entry as an argument, as in the following example:

// unmodifiable Map.Entry containing the given key and value  
import static java.util.Map.entry;  
...  
Map<Integer, MutableMelon> mapOfMelon = Map.ofEntries(  
 entry(1, new MutableMelon("Apollo", 3000)),  
 entry(2, new MutableMelon("Jade Dew", 3500)),  
 entry(3, new MutableMelon("Cantaloupe", 1500))  
);Copy

Alternatively, an immutable Map is another option:

Map<Integer, ImmutableMelon> mapOfMelon = Map.ofEntries(  
 entry(1, new ImmutableMelon("Apollo", 3000)),  
 entry(2, new ImmutableMelon("Jade Dew", 3500)),  
 entry(3, new ImmutableMelon("Cantaloupe", 1500))  
);Copy

In addition, an unmodifiable/immutable Map can be obtained from a modifiable/mutable Map via JDK 10, the Map.copyOf​(Map<? extends K,​? extends V> map) method:

Map<Integer, ImmutableMelon> mapOfMelon = new HashMap<>();  
mapOfMelon.put(1, new ImmutableMelon("Apollo", 3000));  
mapOfMelon.put(2, new ImmutableMelon("Jade Dew", 3500));  
mapOfMelon.put(3, new ImmutableMelon("Cantaloupe", 1500));  
  
Map<Integer, ImmutableMelon> immutableMapOfMelon   
 = Map.copyOf(mapOfMelon);Copy

By way of a bonus for this section, let's talk about an immutable array.

**Question**: Can I create an immutable array in Java?

**Answer**: No, you cannot. Or... there is one way to make an immutable array in Java:

static final String[] immutable = new String[0];Copy

So, all useful arrays in Java are mutable. But we can create a helper class to create immutable arrays based on Arrays.copyOf(), which copies the elements and creates a new array (behind the scenes, this method relies on System.arraycopy()).

So, our helper class is as follows:

import java.util.Arrays;  
  
public final class ImmutableArray<T> {  
  
 private final T[] array;  
  
 private ImmutableArray(T[] a) {  
 array = Arrays.copyOf(a, a.length);  
 }  
  
 public static <T> ImmutableArray<T> from(T[] a) {  
 return new ImmutableArray<>(a);  
 }  
  
 public T get(int index) {  
 return array[index];  
 }  
  
 // equals(), hashCode() and toString() omitted for brevity  
}Copy

A usage example is as follows:

ImmutableArray<String> sample =  
 ImmutableArray.from(new String[] {  
 "a", "b", "c"  
 });Copy

110. Mapping a default value

Before JDK 8, the solution to this problem relied on a helper method, which basically checks the presence of the given key in a Map and returns the corresponding value, or a default value. Such a method can be written in a utility class or by extending the Map interface. By returning a default value, we avoid returning null if the given key was not found in the Map. Moreover, this is a convenient approach for relying on a default setting or configuration.

Starting with JDK 8, the solution to this problem consists of a simple invocation of the Map.getOrDefault() method. This method gets two arguments representing the key to look up in the Map method and the default value. The default value acts as the backup value that should be returned when the given key is not found.

For example, let's assume the following Map that wraps several databases and their default host:port:

Map<String, String> map = new HashMap<>();  
map.put("postgresql", "127.0.0.1:5432");  
map.put("mysql", "192.168.0.50:3306");  
map.put("cassandra", "192.168.1.5:9042");Copy

And, let's try to see whether this Map contains the default host:port for Derby DB as well:

map.get("derby"); // nullCopy

Since Derby DB is not present in the map, the result will be null. This is not what we want. Actually, when the searched database is not present on the map, we can use MongoDB on 69:89.31.226:27017, which is always available. Now, we can easily shape this behavior as follows:

// 69:89.31.226:27017  
String hp1 = map.getOrDefault("derby", "69:89.31.226:27017");  
  
// 192.168.0.50:3306  
String hp2 = map.getOrDefault("mysql", "69:89.31.226:27017");Copy

This method is convenient for building fluent expressions and avoiding disrupting the code for null checks. Note that returning the default value doesn't mean that this value will be added to the Map. Map remains unmodified.

111. Computing whether absent/present in a map

Sometimes, a Map doesn't contain the exact *out-of-the-box* entry that we need. Moreover, when an entry is absent, returning a default entry is not an option as well. Basically, there are cases when we need to compute our entry.

For such cases, JDK 8 comes with a bunch of methods: compute(), computeIfAbsent(), computeIfPresent(), and merge(). Choosing between these methods is a matter of knowing each of them very well.

Let's now take a look at the implementation of these methods using examples.

Example 1 (computeIfPresent())

Let's suppose that we have the following Map:

Map<String, String> map = new HashMap<>();  
map.put("postgresql", "127.0.0.1");  
map.put("mysql", "192.168.0.50");Copy

We use this map to build JDBC URLs for different database types.

Let's assume that we want to build the JDBC URL for MySQL. If the mysql key is present in the map, then the JDBC URL should be computed based on the corresponding value, jdbc:mysql://192.168.0.50/customers\_db. But if the mysql key is not present, then the JDBC URL should be null. In addition to this, if the result of our computation is null (the JDBC URL cannot be computed), then we want to remove this entry from the map.

This is a job for V computeIfPresent​(K key, BiFunction<? super K,​? super V,​? extends V> remappingFunction).

In our case, BiFunction used for computing the new value will be as follows (k is the key from the map, v is the value associated with the key):

BiFunction<String, String, String> jdbcUrl   
 = (k, v) -> "jdbc:" + k + "://" + v + "/customers\_db";Copy

Once we have this function in place, we can compute the new value for the mysql key as follows:

// jdbc:mysql://192.168.0.50/customers\_db  
String mySqlJdbcUrl = map.computeIfPresent("mysql", jdbcUrl);Copy

Since the mysql key is present in the map, the result will be jdbc:mysql://192.168.0.50/customers\_db, and the new map contains the following entries:

postgresql=127.0.0.1, mysql=jdbc:mysql://192.168.0.50/customers\_dbCopy

Calling computeIfPresent() again will recompute the value, which means that it will result in something like mysql= jdbc:mysql://jdbc:mysql://.... Obviously, this is not OK, so pay attention to this aspect.

On the other hand, if we try the same computation for an entry that doesn't exist (for example, voltdb), then the returned value will be null and the map remains untouched:

// null  
String voldDbJdbcUrl = map.computeIfPresent("voltdb", jdbcUrl);Copy

Example 2 (computeIfAbsent())

Let's suppose that we have the following Map:

Map<String, String> map = new HashMap<>();  
map.put("postgresql", "jdbc:postgresql://127.0.0.1/customers\_db");  
map.put("mysql", "jdbc:mysql://192.168.0.50/customers\_db");Copy

We use this map to build JDBC URLs for different databases.

Let's assume that we want to build the JDBC URL for MongoDB. This time, if the mongodb key is present in the map, then the corresponding value should be returned without further computations. But if this key is absent (or is associated with a null value), then it should be computed based on this key and the current IP and be added to the map. If the computed value is null, then null is the returned result and the map remains untouched.

Well, this is a job for V computeIfAbsent​(K key, Function<? super K,​? extends V> mappingFunction).

In our case, Function used to compute the value will be as follows (the first String is the key from the map (k), while the second String is the value computed for this key):

String address = InetAddress.getLocalHost().getHostAddress();  
  
Function<String, String> jdbcUrl   
 = k -> k + "://" + address + "/customers\_db";Copy

Based on this function, we can try to obtain the JDBC URL for MongoDB via the mongodb key as follows:

// mongodb://192.168.100.10/customers\_db  
String mongodbJdbcUrl = map.computeIfAbsent("mongodb", jdbcUrl);Copy

Since our map doesn't contain the mongodb key, it will be computed and added to the map.

If our Function is evaluated to null, then the map remains untouched and the returned value is null.

Calling computeIfAbsent() again will not recompute the value. This time, since mongodb is in the map (it was added at the previous call), the returned value will be mongodb://192.168.100.10/customers\_db. This is the same as trying to fetch the JDBC URL for mysql, which will return jdbc:mysql://192.168.0.50/customers\_db without further computations.

Example 3 (compute())

Let's suppose that we have the following Map:

Map<String, String> map = new HashMap<>();  
map.put("postgresql", "127.0.0.1");  
map.put("mysql", "192.168.0.50");Copy

We use this map to build JDBC URLs for different database types.

Let's assume that we want to build the JDBC URLs for MySQL and Derby DB. In this case, irrespective of whether the key (mysql or derby) is present in the map, the JDBC URL should be computed based on the corresponding key and value (which can be null). In addition, if the key is present in the map and the result of our computation is null (the JDBC URL cannot be computed), then we want to remove this entry from the map. Basically, this is a combination of computeIfPresent() and computeIfAbsent().

This is a job for V compute​(K key, BiFunction<? super K,​? super V,​? extends V> remappingFunction).

This time, BiFunction should be written to cover the case when the value of the searched key is null:

String address = InetAddress.getLocalHost().getHostAddress();  
BiFunction<String, String, String> jdbcUrl = (k, v)   
 -> "jdbc:" + k + "://" + ((v == null) ? address : v)   
 + "/customers\_db";Copy

Now, let's compute the JDBC URL for MySQL. Since the mysql key is present in the map, the computation will rely on the corresponding value, 192.168.0.50. The result will update the value of the mysql key in the map:

// jdbc:mysql://192.168.0.50/customers\_db  
String mysqlJdbcUrl = map.compute("mysql", jdbcUrl);Copy

In addition, let's compute the JDBC URL for Derby DB. Since the derby key is not present in the map, the computation will rely on the current IP. The result will be added to the map under the derby key:

// jdbc:derby://192.168.100.10/customers\_db  
String derbyJdbcUrl = map.compute("derby", jdbcUrl);Copy

After these two computations, the map will contain the following three entries:

* postgresql=127.0.0.1
* derby=jdbc:derby://192.168.100.10/customers\_db
* mysql=jdbc:mysql://192.168.0.50/customers\_db

Pay attention to the fact that calling compute() again will recompute the values. This can lead to unwanted results such as jdbc:derby://jdbc:derby://....  
If the result of the computation is null (for example, the JDBC URL cannot be computed) and the key (for example, mysql) exists in the map, then this entry will be removed from the map and the returned result is null.

Example 4 (merge())

Let's suppose that we have the following Map:

Map<String, String> map = new HashMap<>();  
map.put("postgresql", "9.6.1 ");  
map.put("mysql", "5.1 5.2 5.6 ");Copy

We use this map to store the versions of each database type separated by a space.

Now, let's assume that every time a new version of a database type is released, we want to add it to our map under the corresponding key. If the key (for example, mysql) is present in the map, then we want to simply concatenate the new version to the end of the current value. If the key (for example, derby) is not present in the map, then we just want to add it now.

This is the perfect job for V merge​(K key, V value, BiFunction<? super V,​? super V,​? extends V> remappingFunction).

If the given key (K) is not associated with a value or is associated with null, then the new value will be V. If the given key (K) is associated with a non-null value, then the new value is computed based on the given BiFunction. If the result of this BiFunction is null, and the key is present in the map, then this entry will be removed from the map.

In our case, we want to concatenate the current value with the new version, so our BiFunction can be written as follows:

BiFunction<String, String, String> jdbcUrl = String::concat;Copy

We have a similar situation with the following:

BiFunction<String, String, String> jdbcUrl   
 = (vold, vnew) -> vold.concat(vnew);Copy

For example, let's suppose that we want to concatenate in the map version 8.0 of MySQL. This can be accomplished as follows:

// 5.1 5.2 5.6 8.0  
String mySqlVersion = map.merge("mysql", "8.0 ", jdbcUrl);Copy

Later on, we concatenate version 9.0 as well:

// 5.1 5.2 5.6 8.0 9.0  
String mySqlVersion = map.merge("mysql", "9.0 ", jdbcUrl);Copy

Or, we add version 10.11.1.1 of Derby DB. This will result in a new entry in the map since there is no derby key present:

// 10.11.1.1  
String derbyVersion = map.merge("derby", "10.11.1.1 ", jdbcUrl);Copy

At the end of these three operations, the map entries will be as follows:

postgresql=9.6.1, derby=10.11.1.1, mysql=5.1 5.2 5.6 8.0 9.0Copy

Example 5 (putIfAbsent())

Let's suppose that we have the following Map:

Map<Integer, String> map = new HashMap<>();  
map.put(1, "postgresql");  
map.put(2, "mysql");  
map.put(3, null);Copy

We use this map to store the names of some database types.

Now, let's suppose that we want to include more database types in this map based on the following constraints:

* If the given key is present in the map, then simply return the corresponding value and leave the map untouched.
* If the given key is not present in the map (or is associated with a null value), then put the given value in the map and return null.

Well, this is a job for putIfAbsent​(K key, V value).

The following three attempts speak for themselves:

String v1 = map.putIfAbsent(1, "derby"); // postgresql  
String v2 = map.putIfAbsent(3, "derby"); // null  
String v3 = map.putIfAbsent(4, "cassandra"); // nullCopy

And the map content is as follows:

1=postgresql, 2=mysql, 3=derby, 4=cassandraCopy

112. Removal from a Map

Removal from a Map can be accomplished by a key, or by a key and value.

For example, let's assume that we have the following Map:

Map<Integer, String> map = new HashMap<>();  
map.put(1, "postgresql");  
map.put(2, "mysql");  
map.put(3, "derby");Copy

Removal by key is as simple as calling the V Map.remove(Object key) method. If the entry corresponding to the given key is successfully removed, then this method returns the associated value, otherwise it returns null.

Check the following examples:

String r1 = map.remove(1); // postgresql  
String r2 = map.remove(4); // nullCopy

Now, the map contains the following entries (the entry from key 1 was removed):

2=mysql, 3=derbyCopy

Starting with JDK 8, the Map interface was enriched with a new remove() flag method with the following signature: boolean remove​(Object key, Object value). Using this method, we can remove an entry from a map only if there is a perfect match between the given key and value. Basically, this method is a shortcut of the following compound condition: map.containsKey(key) && Objects.equals(map.get(key), value).

Let's have two simple examples:

// true  
boolean r1 = map.remove(2, "mysql");  
  
// false (the key is present, but the values don't match)  
boolean r2 = map.remove(3, "mysql");Copy

The resultant map contains the single remaining entry, 3=derby.

Iterating and removing from a Map can be accomplished in at least two ways; first, via an Iterator (solution present in the bundled code), and second, starting with JDK 8, we can do it via removeIf​(Predicate<? super E> filter):

map.entrySet().removeIf(e -> e.getValue().equals("mysql"));Copy

More details about removing from a collection are available in the *Removing all elements of a collection that match a predicate* section.

113. Replacing entries from a Map

Replacing entries from a Map is a problem that can be encountered in a wide range of cases. The convenient solution to accomplish this and avoid a snippet of *spaghetti* code written in a helper method relies on JDK 8, the replace() method.

Let's assume that we have the following Melon class and a map of Melon:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals(), hashCode(),  
 // toString() omitted for brevity  
}  
  
Map<Integer, Melon> mapOfMelon = new HashMap<>();  
mapOfMelon.put(1, new Melon("Apollo", 3000));  
mapOfMelon.put(2, new Melon("Jade Dew", 3500));  
mapOfMelon.put(3, new Melon("Cantaloupe", 1500));Copy

Replacing the melon corresponding to key 2 can be accomplished by means of V replace​(K key, V value). If the replacement is successful, then this method will return the initial Melon:

// Jade Dew(3500g) was replaced  
Melon melon = mapOfMelon.replace(2, new Melon("Gac", 1000));Copy

Now, the map contains the following entries:

1=Apollo(3000g), 2=Gac(1000g), 3=Cantaloupe(1500g)Copy

Furthermore, let's suppose that we want to replace the entry with key 1 and the Apollo melon (3,000g). So, the melon should be the same one in order to obtain a successful replacement. This can be accomplished via the Boolean, replace​(K key, V oldValue, V newValue). This method relies on the equals() contract to compare the given values; therefore Melon needs to implement the equals() method, otherwise the result will be unpredictable:

// true  
boolean melon = mapOfMelon.replace(  
 1, new Melon("Apollo", 3000), new Melon("Bitter", 4300));Copy

Now, the map contains the following entries:

1=Bitter(4300g), 2=Gac(1000g), 3=Cantaloupe(1500g)Copy

Finally, let's assume that we want to replace all entries from a Map based on a given function. This can be done via void replaceAll​(BiFunction<? super K,​? super V,​? extends V> function).

For example, let's replace all melons that weigh more than 1,000 g with melons weighing equal to 1,000g. The following BiFunction shapes this function (k is the key and v is the value of each entry from the Map):

BiFunction<Integer, Melon, Melon> function = (k, v)   
 -> v.getWeight() > 1000 ? new Melon(v.getType(), 1000) : v;Copy

Next, replaceAll() appears on the scene:

mapOfMelon.replaceAll(function);Copy

Now, the map contains the following entries:

1=Bitter(1000g), 2=Gac(1000g), 3=Cantaloupe(1000g)Copy

114. Comparing two maps

Comparing two maps is straightforward as long as we rely on the Map.equals() method. When comparing two maps, this method compares the keys and values of them using the Object.equals() method.

For example, let's consider two maps of melons having the same entries (the presence of equals() and hashCode() is a must in the Melon class):

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals(), hashCode(),  
 // toString() omitted for brevity  
}  
  
Map<Integer, Melon> melons1Map = new HashMap<>();  
Map<Integer, Melon> melons2Map = new HashMap<>();  
melons1Map.put(1, new Melon("Apollo", 3000));  
melons1Map.put(2, new Melon("Jade Dew", 3500));  
melons1Map.put(3, new Melon("Cantaloupe", 1500));  
melons2Map.put(1, new Melon("Apollo", 3000));  
melons2Map.put(2, new Melon("Jade Dew", 3500));  
melons2Map.put(3, new Melon("Cantaloupe", 1500));Copy

Now, if we test melons1Map and melons2Map for equality, then we obtain true:

boolean equals12Map = melons1Map.equals(melons2Map); // trueCopy

But this will not work if we use arrays. For example, consider the next two maps:

Melon[] melons1Array = {  
 new Melon("Apollo", 3000),  
 new Melon("Jade Dew", 3500), new Melon("Cantaloupe", 1500)  
};  
Melon[] melons2Array = {  
 new Melon("Apollo", 3000),  
 new Melon("Jade Dew", 3500), new Melon("Cantaloupe", 1500)  
};  
  
Map<Integer, Melon[]> melons1ArrayMap = new HashMap<>();  
melons1ArrayMap.put(1, melons1Array);  
Map<Integer, Melon[]> melons2ArrayMap = new HashMap<>();  
melons2ArrayMap.put(1, melons2Array);Copy

Even if melons1ArrayMap and melons2ArrayMap are equal, Map.equals() will return false:

boolean equals12ArrayMap = melons1ArrayMap.equals(melons2ArrayMap);Copy

The problem originates in the fact that the array's equals() method compares identity and not the contents of the array. In order to solve this problem, we can write a helper method as follows (this time relying on Arrays.equals(), which compares the contents of the arrays):

public static <A, B> boolean equalsWithArrays(  
 Map<A, B[]> first, Map<A, B[]> second) {  
  
 if (first.size() != second.size()) {  
 return false;  
 }  
  
 return first.entrySet().stream()  
 .allMatch(e -> Arrays.equals(e.getValue(),   
 second.get(e.getKey())));  
}Copy

115. Sorting a Map

There are several solutions for sorting a Map. For a start, let's assume the following Map of Melon:

public class Melon implements Comparable {  
  
 private final String type;  
 private final int weight;  
  
 @Override  
 public int compareTo(Object o) {  
 return Integer.compare(this.getWeight(), ((Melon) o).getWeight());  
 }  
  
 // constructor, getters, equals(), hashCode(),  
 // toString() omitted for brevity  
}  
  
Map<String, Melon> melons = new HashMap<>();  
melons.put("delicious", new Melon("Apollo", 3000));  
melons.put("refreshing", new Melon("Jade Dew", 3500));  
melons.put("famous", new Melon("Cantaloupe", 1500));Copy

Now, let's examine several solutions for sorting this Map. Basically, the goal is to expose the methods from the following screenshot via a utility class named Maps:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Let's take a look at the different solutions in the next sections.

Sorting by key via TreeMap and natural ordering

A quick solution to sorting a Map relies on TreeMap. By definition, the keys in TreeMap are sorted by their natural order. Moreover, TreeMap has a constructor of the TreeMap​(Map<? extends K,​? extends V> m) type:

public static <K, V> TreeMap<K, V> sortByKeyTreeMap(Map<K, V> map) {  
  
 return new TreeMap<>(map);  
}Copy

And calling it will sort the map by key:

// {delicious=Apollo(3000g),   
// famous=Cantaloupe(1500g), refreshing=Jade Dew(3500g)}  
TreeMap<String, Melon> sortedMap = Maps.sortByKeyTreeMap(melons);Copy

Sorting by key and value via Stream and Comparator

Once we create a Stream for a map, we can easily sort it by means of the Stream.sorted() method with or without a Comparator. This time, let's use a Comparator:

public static <K, V> Map<K, V> sortByKeyStream(  
 Map<K, V> map, Comparator<? super K> c) {  
  
 return map.entrySet()  
 .stream()  
 .sorted(Map.Entry.comparingByKey(c))  
 .collect(toMap(Map.Entry::getKey, Map.Entry::getValue,  
 (v1, v2) -> v1, LinkedHashMap::new));  
}  
  
public static <K, V> Map<K, V> sortByValueStream(  
 Map<K, V> map, Comparator<? super V> c) {  
  
 return map.entrySet()  
 .stream()  
 .sorted(Map.Entry.comparingByValue(c))  
 .collect(toMap(Map.Entry::getKey, Map.Entry::getValue,  
 (v1, v2) -> v1, LinkedHashMap::new));  
}Copy

We need to rely on LinkedHashMap instead of HashMap. Otherwise, we cannot preserve the iteration order.

Let's sort our map as follows:

// {delicious=Apollo(3000g),   
// famous=Cantaloupe(1500g),   
// refreshing=Jade Dew(3500g)}  
Comparator<String> byInt = Comparator.naturalOrder();  
Map<String, Melon> sortedMap = Maps.sortByKeyStream(melons, byInt);  
  
// {famous=Cantaloupe(1500g),   
// delicious=Apollo(3000g),   
// refreshing=Jade Dew(3500g)}  
Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);  
Map<String, Melon> sortedMap   
 = Maps.sortByValueStream(melons, byWeight);Copy

Sorting by key and value via List

The preceding examples sort the given map, and the result is also a map. If all we need is the sorted keys (and we don't care about the values) or vice versa, then we can rely on a List created via Map.keySet() for keys, and via Map.values() for values:

public static <K extends Comparable, V> List<K>  
 sortByKeyList(Map<K, V> map) {  
  
 List<K> list = new ArrayList<>(map.keySet());  
 Collections.sort(list);  
  
 return list;  
}  
  
public static <K, V extends Comparable> List<V>  
 sortByValueList(Map<K, V> map) {  
  
 List<V> list = new ArrayList<>(map.values());  
 Collections.sort(list);  
  
 return list;  
}Copy

Now, let's sort our map:

// [delicious, famous, refreshing]  
List<String> sortedKeys = Maps.sortByKeyList(melons);  
  
// [Cantaloupe(1500g), Apollo(3000g), Jade Dew(3500g)]  
List<Melon> sortedValues = Maps.sortByValueList(melons);Copy

If duplicate values are not allowed, then you have to rely on an implementation using SortedSet:

SortedSet<String> sortedKeys = new TreeSet<>(melons.keySet());  
SortedSet<Melon> sortedValues = new TreeSet<>(melons.values());Copy

116. Copying HashMap

A handy solution for performing a shallow copy of HashMap relies on the HashMap constructor, HashMap​(Map<? extends K,​? extends V> m). The following code is self-explanatory:

Map<K, V> mapToCopy = new HashMap<>();  
Map<K, V> shallowCopy = new HashMap<>(mapToCopy);Copy

Another solution may rely on the putAll​(Map<? extends K,​? extends V> m) method. This method copies all of the mappings from the specified map to this map, as shown in the following helper method:

@SuppressWarnings("unchecked")  
public static <K, V> HashMap<K, V> shallowCopy(Map<K, V> map) {  
  
 HashMap<K, V> copy = new HashMap<>();  
 copy.putAll(map);  
  
 return copy;  
}Copy

We can also write a helper method in Java 8 functional style as follows:

@SuppressWarnings("unchecked")  
public static <K, V> HashMap<K, V> shallowCopy(Map<K, V> map) {  
  
 Set<Entry<K, V>> entries = map.entrySet();  
 HashMap<K, V> copy = (HashMap<K, V>) entries.stream()  
 .collect(Collectors.toMap(  
 Map.Entry::getKey, Map.Entry::getValue));  
  
 return copy;  
}Copy

However, these three solutions only provide a shallow copy of the map. A solution for obtaining a deep copy can rely on the Cloning library (<https://github.com/kostaskougios/cloning>) introduced in [Chapter 2](https://subscription.packtpub.com/book/programming/9781789801415/5/ch05lvl1sec20/16027e66-3bab-4bff-b106-7cbe15480be6.xhtml), *Objects, Immutability, and Switch Expressions*. A helper method that will use Cloning can be written as follows:

@SuppressWarnings("unchecked")   
public static <K, V> HashMap<K, V> deepCopy(Map<K, V> map) {  
 Cloner cloner = new Cloner();  
 HashMap<K, V> copy = (HashMap<K, V>) cloner.deepClone(map);  
  
 return copy;  
}Copy

117. Merging two maps

Merging two maps is the process of joining two maps into a single map that contains the elements of both maps. Furthermore, for key collisions, we incorporate in the final map the value belonging to the second map. But this is a design decision.

Let's consider the following two maps (we intentionally added a collision for key 3):

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals(), hashCode(),  
 // toString() omitted for brevity  
}  
  
Map<Integer, Melon> melons1 = new HashMap<>();  
Map<Integer, Melon> melons2 = new HashMap<>();  
melons1.put(1, new Melon("Apollo", 3000));  
melons1.put(2, new Melon("Jade Dew", 3500));  
melons1.put(3, new Melon("Cantaloupe", 1500));  
melons2.put(3, new Melon("Apollo", 3000));  
melons2.put(4, new Melon("Jade Dew", 3500));  
melons2.put(5, new Melon("Cantaloupe", 1500));Copy

Starting with JDK 8, we have the following method in Map: V merge​(K key, V value, BiFunction<? super V,​? super V,​? extends V> remappingFunction).

If the given key (K) is not associated with a value, or is associated with null, then the new value will be V. If the given key (K) is associated with a non-null value, then the new value is computed based on the given BiFunction. If the result of this BiFunction is null, and the key is present in the map, then this entry will be removed from the map.

Based on this definition, we can write a helper method for merging two maps as follows:

public static <K, V> Map<K, V> mergeMaps(  
 Map<K, V> map1, Map<K, V> map2) {   
  
 Map<K, V> map = new HashMap<>(map1);  
  
 map2.forEach(  
 (key, value) -> map.merge(key, value, (v1, v2) -> v2));  
  
 return map;  
}Copy

Note that we don't modify the original maps. We prefer to return a new map containing the elements of the first map merged with the elements of the second map. In the case of a collision of keys, we replace the existing value with the value from the second map (v2).

Another solution can be written based on Stream.concat(). Basically, this method concatenates two streams into a single Stream. In order to create a Stream from a Map, we call Map.entrySet().stream(). After concatenating the two streams created from the given maps, we simply collect the result via the toMap() collector:

public static <K, V> Map<K, V> mergeMaps(  
 Map<K, V> map1, Map<K, V> map2) {  
  
 Stream<Map.Entry<K, V>> combined   
 = Stream.concat(map1.entrySet().stream(),   
 map2.entrySet().stream());  
  
 Map<K, V> map = combined.collect(  
 Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,  
 (v1, v2) -> v2));  
  
 return map;  
}Copy

As a bonus, a Set (for example, a Set of integers) can be sorted as follows:

List<Integer> sortedList = someSetOfIntegers.stream()  
 .sorted().collect(Collectors.toList());Copy

For objects, rely on sorted(Comparator<? super T>.

118. Removing all elements of a collection that match a predicate

Our collection will hold a bunch of Melon:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals(),   
 // hashCode(), toString() omitted for brevity  
}Copy

Let's assume the following collection (ArrayList) throughout our examples to demonstrate how we can remove elements from it that match a given predicate:

List<Melon> melons = new ArrayList<>();  
melons.add(new Melon("Apollo", 3000));  
melons.add(new Melon("Jade Dew", 3500));  
melons.add(new Melon("Cantaloupe", 1500));  
melons.add(new Melon("Gac", 1600));  
melons.add(new Melon("Hami", 1400));Copy

Let's take a look at the different solutions given in the following sections.

Removing via an iterator

Removing via an Iterator is the oldest approach available in Java. Mainly, an Iterator allows us to iterate (or traverse) a collection and remove certain elements. The oldest approach also has some drawbacks. First of all, depending on the collection type, removing via an Iterator is prone to ConcurrentModificationException if multiple threads modify the collection. Moreover, removal does not behave the same for all collections (for example, removing from a LinkedList is faster than removing from an ArrayList because the former simply moves the pointer to the next element while the latter needs to shift elements). Nevertheless, the solution is available in the bundled code.

If all you need is the size of Iterable, then consider one of the following approaches:

// for any Iterable  
StreamSupport.stream(*iterable*.spliterator(), false).count();  
  
// for collections  
((Collection<?>) *iterable*).size()Copy

Removing via Collection.removeIf()

Starting with JDK 8, we can reduce the preceding code to a single line of code via the Collection.removeIf() method. This method relies on Predicate, as in the following example:

melons.removeIf(t -> t.getWeight() < 3000);Copy

This time, the ArrayList iterates the list and marks for deletion those elements that satisfy our Predicate. Furthermore, ArrayList iterates again to remove the marked elements and shift the remaining elements.

Using this approach, LinkedList and ArrayList perform in almost an identical fashion.

Removing via Stream

Starting with JDK 8, we can create a Stream from a collection (Collection.stream()) and filter its elements via filter(Predicate p). The filter will only retain those elements that satisfy the given Predicate.

Finally, we collect these elements via the proper collector:

List<Melon> filteredMelons = melons.stream()  
 .filter(t -> t.getWeight() >= 3000)  
 .collect(Collectors.toList());Copy

Unlike the other two solutions, this one doesn't mutate the original collection, but it may be slower and consume more memory.

Separating elements via Collectors.partitioningBy()

Sometimes, we don't want to delete the elements that don't match our predicate. What we actually want is to separate elements based on our predicate. Well, this is achievable via Collectors.partitioningBy(Predicate p).

Basically, Collectors.partitioningBy() will separate the elements into two lists. These two lists are added to a Map as values. The two keys of this Map will be true and false:

Map<Boolean, List<Melon>> separatedMelons = melons.stream()  
 .collect(Collectors.partitioningBy(  
 (Melon t) -> t.getWeight() >= 3000));  
  
List<Melon> weightLessThan3000 = separatedMelons.get(false);  
List<Melon> weightGreaterThan3000 = separatedMelons.get(true);Copy

So, the true key is for retrieving the List that contains the elements that match the predicate, while the false key is for retrieving the List that contains the elements that didn't match the predicate.

By way of a bonus, if we want to check whether all the elements of a List are the same, then we can rely on Collections.frequency(Collection c, Object obj). This method returns the number of elements in the specified collection equal to the specified object:

boolean allTheSame = Collections.frequency(  
 melons, melons.get(0)) == melons.size());Copy

If allTheSame is true, then all elements are the same. Note that equals() and hashCode() of the object from the List must be implemented accordingly.

119. Converting a collection into an array

In order to convert a collection into an array, we can rely on the Collection.toArray() method. Without arguments, this method will convert the given collection into an Object[], as in the following example:

List<String> names = Arrays.asList("ana", "mario", "vio");  
Object[] namesArrayAsObjects = names.toArray();Copy

Obviously, this is not entirely useful since we are expecting a String[] instead of Object[]. This can be accomplished via Collection.toArray​(T[] a) as follows:

String[] namesArraysAsStrings = names.toArray(new String[names.size()]);  
String[] namesArraysAsStrings = names.toArray(new String[0]);Copy

From these two solutions, the second one is preferable since we avoid computing the collection size.

But starting with JDK 11, there is one more method dedicated to this task, Collection.toArray​(IntFunction<T[]> generator). This method returns an array containing all the elements in this collection, using the generator function provided to allocate the returned array:

String[] namesArraysAsStrings = names.toArray(String[]::new);Copy

Next to the fixed-size modifiable Arrays.asList(), we can build an unmodifiable List/Set from an array via the of() methods:

String[] namesArray = {"ana", "mario", "vio"};  
  
List<String> namesArrayAsList = List.of(namesArray);  
Set<String> namesArrayAsSet = Set.of(namesArray);Copy

120. Filtering a Collection by a List

A common problem that we encounter in applications is filtering a Collection by a List. Mainly, we start from a huge Collection, and we want to extract from it the elements that match the elements of a List.

In the following examples, let's consider the Melon class:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals(), hashCode(),  
 // toString() omitted for brevity  
}Copy

Here, we have a huge Collection (in this case, an ArrayList) of Melon:

List<Melon> melons = new ArrayList<>();  
melons.add(new Melon("Apollo", 3000));  
melons.add(new Melon("Jade Dew", 3500));  
melons.add(new Melon("Cantaloupe", 1500));  
melons.add(new Melon("Gac", 1600));  
melons.add(new Melon("Hami", 1400));  
...Copy

And we also have a List containing the types of melons that we want to extract from the preceding ArrayList:

List<String> melonsByType   
 = Arrays.asList("Apollo", "Gac", "Crenshaw", "Hami");Copy

One solution to this problem may involve looping both collections and comparing the types of melons, but the resultant code will be pretty slow. Another solution to this problem may involve the List.contains() method and a lambda expression:

List<Melon> results = melons.stream()  
 .filter(t -> melonsByType.contains(t.getType()))  
 .collect(Collectors.toList());Copy

The code is compact and fast. Behind the scenes, List.contains() relies on the following check:

// size - the size of melonsByType  
// o - the current element to search from melons  
// elementData - melonsByType  
for (int i = 0; i < size; i++)  
 if (o.equals(elementData[i])) {  
 return i;  
 }  
}Copy

However, we can give another boost to performance via a solution that relies on HashSet.contains() instead of List.contains(). While List.contains() uses the preceding for statement to match the elements, HashSet.contains() uses Map.containsKey(). Mainly, Set is implemented based on a Map, and each added element is mapped as a key-value of the *element*-PRESENT type. So, *element* is a key in this Map, while PRESENT is just a dummy value.

When we call HashSet.contains(element), we actually call Map.containsKey(element). This method matches the given element with the proper key in the map based on its hashCode(), which is much faster than equals().

Once we convert the initial ArrayList to a HashSet, we are ready to go:

Set<String> melonsSetByType = melonsByType.stream()  
 .collect(Collectors.toSet());  
  
List<Melon> results = melons.stream()  
 .filter(t -> melonsSetByType.contains(t.getType()))  
 .collect(Collectors.toList());Copy

Well, this solution is faster than the previous one. It should run in half of the time required by the previous solution.

121. Replacing elements of a List

Another common problem that we encounter in applications entails replacing the elements of a List that matches certain conditions.

In the following example, let's consider the Melon class:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals(), hashCode(),  
 // toString() omitted for brevity  
}Copy

And then, let's consider a List of Melon:

List<Melon> melons = new ArrayList<>();  
  
melons.add(new Melon("Apollo", 3000));  
melons.add(new Melon("Jade Dew", 3500));  
melons.add(new Melon("Cantaloupe", 1500));  
melons.add(new Melon("Gac", 1600));  
melons.add(new Melon("Hami", 1400));Copy

Let's assume that we want to replace all melons weighing less than 3,000 grams with other melons of the same types and that weigh 3,000 grams.

A solution to this problem will entail iterating the List and then using List.set(int index, E element) to replace the melons accordingly.

This is a snippet of spaghetti code as follows:

for (int i = 0; i < melons.size(); i++) {  
  
 if (melons.get(i).getWeight() < 3000) {  
  
 melons.set(i, new Melon(melons.get(i).getType(), 3000));  
 }  
}Copy

Another solution relies on Java 8 functional style or, more precisely, on the UnaryOperator functional interface.

Based on this functional interface, we can write the following operator:

UnaryOperator<Melon> operator = t   
 -> (t.getWeight() < 3000) ? new Melon(t.getType(), 3000) : t;Copy

Now, we can use the JDK 8, List.replaceAll(UnaryOperator<E> operator), as follows:

melons.replaceAll(operator);Copy

Both approaches should perform almost the same.

122. Thread-safe collections, stacks, and queues

Whenever a collection/stack/queue is prone to be accessed by multiple threads, it is also prone to concurrency-specific exceptions (for example, java.util.ConcurrentModificationException). Now, let's have a brief overview of, and introduction to, the Java built-in concurrent collections.

Concurrent collections

Fortunately, Java provides thread-safe (concurrent) alternatives to non-thread-safe collections (including stacks and queues), as follows.

Thread-safe lists

The thread-safe version of an ArrayList is CopyOnWriteArrayList. The following table enumerates the Java built-in single-threaded and multithreaded lists:

|  |  |
| --- | --- |
| **Single thread** | **Multithreaded** |
| ArrayList  LinkedList | CopyOnWriteArrayList (often reads, seldom updates)  Vector |

The CopyOnWriteArrayList implementation holds the elements in an array. Every time we invoke a method that mutates the list (for example, add(), set(), and remove()), Java will operate on a copy of this array.

An Iterator over this collection will operate on an immutable copy of the collection. Therefore, the original collection can be modified without issues. Potential modifications of the original collection are not visible in the Iterator:

List<Integer> list = new CopyOnWriteArrayList<>();Copy

Use this collection when reads are frequent and changes are seldom.

Thread-safe set

The thread-safe version of a Set is CopyOnWriteArraySet. The following table enumerates the Java built-in single-threaded and multithreaded sets:

|  |  |
| --- | --- |
| **Single thread** | **Multithreaded** |
| HashSet  TreeSet (sorted set)  LinkedHashSet (maintain insertions order)  BitSet  EnumSet | ConcurrentSkipListSet (sorted set)  CopyOnWriteArraySet (often reads, seldom updates) |

This is a Set that uses an internal CopyOnWriteArrayList for all of its operations. Creating such a Set can be done as follows:

Set<Integer> set = new CopyOnWriteArraySet<>();Copy

Use this collection when reads are frequent and changes are seldom.

The thread-safe version of NavigableSet is ConcurrentSkipListSet (concurrent SortedSet implementation, with most basic operations in O(log n)).

Thread-safe map

The thread-safe version of a Map is ConcurrentHashMap.

The following table enumerates the Java built-in single-thread and multithreaded maps:

|  |  |
| --- | --- |
| **Single thread** | **Multithreaded** |
| HashMap  TreeMap (sorted keys)  LinkedHashMap (maintain insertion order)  IdentityHashMap (keys compared via ==)  WeakHashMap  EnumMap | ConcurrentHashMap  ConcurrentSkipListMap (sorted map)  Hashtable |

ConcurrentHashMap allows retrieval operations (for example, get()) without blocking. This means that retrieval operations may overlap with update operations (including put() and remove()).

Creating a ConcurrentHashMap can be done as follows:

ConcurrentMap<Integer, Integer> map = new ConcurrentHashMap<>();Copy

Whenever thread safety and high performance are required, you can rely on the thread-safe version of a Map , which is ConcurrentHashMap.  
  
Avoid Hashtable and Collections.synchronizedMap() since they have poor performance.

For a ConcurrentMap supporting NavigableMap, operations rely on ConcurrentSkipListMap:

ConcurrentNavigableMap<Integer, Integer> map   
 = new ConcurrentSkipListMap<>();Copy

Thread-safe queue backed by an array

Java provides a thread-safe queue (**First In First Out**(**FIFO**)) backed by an array via ArrayBlockingQueue. The following table lists the single-thread and multithreaded Java built-in queues backed by an array:

|  |  |
| --- | --- |
| **Single thread** | **Multithreaded** |
| ArrayDeque  PriorityQueue (sorted retrievals) | ArrayBlockingQueue (bounded)  ConcurrentLinkedQueue (unbounded)  ConcurrentLinkedDeque (unbounded)  LinkedBlockingQueue (optionally bounded)  LinkedBlockingDeque (optionally bounded)  LinkedTransferQueue  PriorityBlockingQueue  SynchronousQueue  DelayQueue  Stack |

The capacity of ArrayBlockingQueue cannot be changed following creation. Attempts to put an element into a full queue will result in the operation blocking; attempts to take an element from an empty queue will similarly block.

Creating ArrayBlockingQueue can easily be done as follows:

BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(QUEUE\_MAX\_SIZE);Copy

Java also comes with two thread-safe, optionally bounded blocking queues based on linked nodes via LinkedBlockingQueue and LinkedBlockingDeque (a deque is a linear collection that supports element insertion and removal at both ends).

Thread-safe queue based on linked nodes

Java provides an unbounded thread-safe queue/deque backed by linked nodes via ConcurrentLinkedDeque/ ConcurrentLinkedQueue. Here, it is ConcurrentLinkedDeque:

Deque<Integer> queue = new ConcurrentLinkedDeque<>();Copy

Thread-safe priority queue

Java provides an unbounded thread-safe priority blocking queue based on a priority heap via PriorityBlockingQueue.

Creating PriorityBlockingQueue can easily be done as follows:

BlockingQueue<Integer> queue = new PriorityBlockingQueue<>();Copy

The non-thread-safe version is named PriorityQueue.

Thread-safe delay queue

Java provides a thread-safe unbounded blocking queue in which an element can only be taken when its delay has expired via DelayQueue. Creating a DelayQueue is as simple as the following:

BlockingQueue<TrainDelay> queue = new DelayQueue<>();Copy

Thread-safe transfer queue

Java provides a thread-safe unbounded transfer queue based on linked nodes via LinkedTransferQueue.

This is a FIFO queue in which the *head* is the element that has been on the queue the longest time for some producer. The *tail* of the queue is the element that has been on the queue the shortest time for some producer.

One way to create this kind of queue is as follows:

TransferQueue<String> queue = new LinkedTransferQueue<>();Copy

Thread-safe synchronous queue

Java provides a blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa, via SynchronousQueue:

BlockingQueue<String> queue = new SynchronousQueue<>();Copy

Thread-safe stack

Thread-safe implementations of a stack are Stack and ConcurrentLinkedDeque.

The Stack class represents a **Last In First Out** (**LIFO**) stack of objects. It extends the Vector class with several operations that allow a vector to be treated as a stack. Every method of Stack is synchronized. Creating a Stack is as simple as the following:

Stack<Integer> stack = new Stack<>();Copy

A ConcurrentLinkedDeque implementation can be used as a Stack (LIFO) via its push() and pop() methods:

Deque<Integer> stack = new ConcurrentLinkedDeque<>();Copy

For better performance, prefer ConcurrentLinkedDeque over Stack.

The code bundled to this book comes with an application for each of the preceding collections meant to span several threads in order to reveal their thread-safe character.

Synchronized collections

Besides concurrent collections, we also have synchronized collections. Java provides a suite of wrappers that expose a collection as a thread-safe collection. These wrappers are available in Collections. The most common ones are as follows:

* synchronizedCollection​(Collection<T> c): Returns a synchronized (thread-safe) collection backed by the specified collection
* synchronizedList​(List<T> list): Returns a synchronized (thread-safe) list backed by the specified list:

List<Integer> syncList   
 = Collections.synchronizedList(new ArrayList<>());Copy

* synchronizedMap​(Map<K,​V> m): Returns a synchronized (thread-safe) map backed by the specified map:

Map<Integer, Integer> syncMap   
 = Collections.synchronizedMap(new HashMap<>());Copy

* synchronizedSet​(Set<T> s): Returns a synchronized (thread-safe) set backed by the specified set:

Set<Integer> syncSet   
 = Collections.synchronizedSet(new HashSet<>());Copy

Concurrent versus synchronized collections

The obvious question is *What is the difference between a concurrent and a synchronized collection?* Well, the main difference consists of the way in which they achieve thread-safety. Concurrent collections achieve thread-safety by partitioning the data into segments. Threads can access these segments concurrently and obtain locks only on the segments that are used. On the other hand, synchronized collection locks the entire collection via *intrinsic locking* (a thread that invokes a synchronized method will automatically acquire the intrinsic lock for that method's object and release it when the method returns).

Iterating a synchronized collection requires manual synchronization as follows:

List syncList = Collections.synchronizedList(new ArrayList());  
...  
synchronized(syncList) {  
 Iterator i = syncList.iterator();  
 while (i.hasNext()) {  
 *do\_something\_with* i.next();  
 }  
}Copy

Since concurrent collections allow concurrent access of threads, they are much more performant than the synchronized collection.

123. Breadth-first search

BFS is a classic algorithm for traversing (visiting) all nodes of a graph or tree.

The easiest way to understand this algorithm is via pseudo-code and an example. The pseudo-code of BFS is as follows:

1. *Create a queue Q*
2. *Mark v as visited and put v into Q*
3. *While* *Q is non-empty*
4. *Remove the head h of Q*
5. *Mark and en-queue all (unvisited) neighbors of h*

Let's assume the graph from the following diagram, **Step 0**:

Diagrama, Esquemático

Descripción generada automáticamente

At the first step (**Step 1**), we visit vertex **0**. We put this in the visited list and all its adjacent vertices in the queue (3, 1). Furthermore, at **Step 2**, we visit the element at the front of the queue, **3**. Vertex **3** has an unvisited adjacent vertex in **2**, so we add that to the back of the queue. Next, at **Step 3**, we visit the element at the front of the queue, **1**. This vertex has a single adjacent vertex (**0**), but this was visited. Finally, we visit vertex **2**, the last from the queue. This one has a single adjacent vertex (**3**) that was already visited.

In code lines, the BFS algorithm can be implemented as follows:

public class Graph {  
  
 private final int v;  
 private final LinkedList<Integer>[] adjacents;  
  
 public Graph(int v) {  
  
 this.v = v;  
 adjacents = new LinkedList[v];  
  
 for (int i = 0; i < v; ++i) {  
 adjacents[i] = new LinkedList();  
 }  
 }  
  
 public void addEdge(int v, int e) {  
 adjacents[v].add(e);  
 }  
  
 public void BFS(int start) {  
  
 boolean visited[] = new boolean[v];  
 LinkedList<Integer> queue = new LinkedList<>();  
 visited[start] = true;  
  
 queue.add(start);  
  
 while (!queue.isEmpty()) {  
 start = queue.poll();  
 System.out.print(start + " ");  
  
 Iterator<Integer> i = adjacents[start].listIterator();  
 while (i.hasNext()) {  
 int n = i.next();  
 if (!visited[n]) {  
 visited[n] = true;  
 queue.add(n);  
 }  
 }  
 }  
 }  
}Copy

And, if we introduce the following graph (from the preceding diagram), we have the following:

Graph graph = new Graph(4);  
graph.addEdge(0, 3);  
graph.addEdge(0, 1);  
graph.addEdge(1, 0);  
graph.addEdge(2, 3);  
graph.addEdge(3, 0);  
graph.addEdge(3, 2);  
graph.addEdge(3, 3);Copy

The output will be 0 3 1 2.

124. Trie

A Trie (also known as digital tree) is an ordered tree structure used commonly for storing strings. Its name comes from the fact that Trie is re*Trie*val data structure. Its performance is better than a binary tree.

Except for the root of the Trie, every node of a Trie contains a single character (for example, for the word **hey**, there will be three nodes). Mainly, each node of a Trie contains the following:

* A value (a character, or a digit)
* Pointers to children nodes
* A flag that is true if the current node completes a word
* A single root used for branching nodes

The following diagram represents the sequence of steps for building a Trie containing the words **cat**, **caret**, and **bye**:

Diagrama

Descripción generada automáticamente

So, in code lines, a Trie node can be shaped as follows:

public class Node {  
  
 private final Map<Character, Node> children = new HashMap<>();  
 private boolean word;  
  
 Map<Character, Node> getChildren() {  
 return children;  
 }  
  
 public boolean isWord() {  
 return word;  
 }  
  
 public void setWord(boolean word) {  
 this.word = word;  
 }  
}Copy

Based on this class, we can define a Trie basic structure as follows:

class Trie {  
  
 private final Node root;  
  
 public Trie() {  
 root = new Node();  
 }  
  
 public void insert(String word) {  
 ...  
 }  
  
 public boolean contains(String word) {  
 ...  
 }  
  
 public boolean delete(String word) {  
 ...  
 }  
}Copy

Inserting in a Trie

Now, let's focus on the algorithm for inserting words in a Trie:

1. Consider the current node as the root.
2. Loop the given word character by character, starting from the first character.
3. If the current node (the Map<Character, Node>) maps a value (a Node) for the current character, then simply advance to this node. Otherwise, create a new Node, set its character equal to the current character, and advance to this node.
4. Repeat from step 2 (pass to next character) until the end of the word.
5. Mark the current node as a node that completes the word.

In terms of code lines, we have the following:

public void insert(String word) {  
  
 Node node = root;  
  
 for (int i = 0; i < word.length(); i++) {  
 char ch = word.charAt(i);  
 Function function = k -> new Node();  
  
 node = node.getChildren().computeIfAbsent(ch, function);  
 }  
  
 node.setWord(true);  
}Copy

The complexity of insertion is O(n), where n represents the word size.

Finding in a Trie

Now, let's search for a word in a Trie:

1. Consider the current node as the root.
2. Loop the given word character by character (start from the first character).
3. For each character, check its presence in the Trie (in Map<Character, Node>).
4. If a character is not present, then return false.
5. Repeat from step 2 until the end of the word.
6. At the end of the word, return true if this was a word, or false if it was just a prefix.

In terms of code lines, we have the following:

public boolean contains(String word) {  
  
 Node node = root;  
  
 for (int i = 0; i < word.length(); i++) {  
 char ch = word.charAt(i);  
 node = node.getChildren().get(ch);  
  
 if (node == null) {  
 return false;  
 }  
 }  
  
 return node.isWord();  
}Copy

The complexity of finding is O(n), where n represents the word size.

Deleting from a Trie

Finally, let's try to delete from a Trie:

1. Verify whether the given word is part of the Trie.
2. If it is part of the Trie, then simply remove it.

Deletion takes place in a bottom-up manner using recursion and following these rules:

* If the given word is not in the Trie, then nothing happens (return false)
* If the given word is unique (not part of another word), then delete all corresponding nodes (return true)
* If the given word is a prefix of another long word in the Trie, then set the leaf node flag to false (return false)
* If the given word has at least another word as a prefix, then delete the corresponding nodes from the end of the given word until the first leaf node of the longest prefix word (return false)

In terms of code lines, we have the following:

public boolean delete(String word) {  
 return delete(root, word, 0);  
}  
  
private boolean delete(Node node, String word, int position) {  
  
 if (word.length() == position) {  
 if (!node.isWord()) {  
 return false;  
 }  
  
 node.setWord(false);  
  
 return node.getChildren().isEmpty();  
 }  
  
 char ch = word.charAt(position);  
 Node children = node.getChildren().get(ch);  
  
 if (children == null) {  
 return false;  
 }  
  
 boolean deleteChildren = delete(children, word, position + 1);  
  
 if (deleteChildren && !children.isWord()) {  
 node.getChildren().remove(ch);  
  
 return node.getChildren().isEmpty();  
 }  
  
 return false;  
}Copy

The complexity of finding is O(n), where n represents the word size.

Now, we can build a Trie as follows:

Trie trie = new Trie();  
trie.insert/contains/delete(...);Copy

125. Tuple

Basically, a tuple is a data structure consisting of multiple parts. Usually, a tuple has two or three parts. Typically, when more than three parts are needed, a dedicated class is a better choice.

Tuples are immutable and are used whenever we need to return multiple results from a method. For example, let's assume that we have a method that returns the minimum and maximum of an array. Normally, a method cannot return both, and using a tuple is a convenient solution.

Unfortunately, Java doesn't provide built-in tuple support. Nevertheless, Java comes with Map.Entry<K,​V>, which is used to represent an entry from a Map. Moreover, starting with JDK 9, the Map interface was enriched with a method named entry(K k, V v), which returns an unmodifiable Map.Entry<K, V> containing the given key and value.

For a tuple of two parts, we can write our method as follows:

public static <T> Map.Entry<T, T> array(  
 T[] arr, Comparator<? super T> c) {  
  
 T min = arr[0];  
 T max = arr[0];  
  
 for (T elem: arr) {  
 if (c.compare(min, elem) > 0) {  
 min = elem;  
 } else if (c.compare(max, elem)<0) {  
 max = elem;  
 }  
 }  
  
 return entry(min, max);  
}Copy

If this method lives in a class named Bounds, then we can call it as follows:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructor, getters, equals(), hashCode(),  
 // toString() omitted for brevity  
}  
  
Melon[] melons = {  
 new Melon("Crenshaw", 2000), new Melon("Gac", 1200),  
 new Melon("Bitter", 2200), new Melon("Hami", 800)  
};  
  
Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);  
Map.Entry<Melon, Melon> minmax = Bounds.array(melons, byWeight);  
  
System.out.println("Min: " + minmax1.getKey()); // Hami(800g)  
System.out.println("Max: " + minmax1.getValue()); // Bitter(2200g)Copy

But we can write an implementation as well. A tuple with two parts is commonly named a *pair*; therefore, an intuitive implementation can be as follows:

public final class Pair<L, R> {  
  
 final L left;  
 final R right;  
  
 public Pair(L left, R right) {  
 this.left = left;  
 this.right = right;  
 }  
  
 static <L, R> Pair<L, R> of (L left, R right) {  
  
 return new Pair<>(left, right);  
 }  
  
 // equals() and hashCode() omitted for brevity  
}Copy

Now, we can rewrite our method that computes the minimum and maximum as follows:

public static <T> Pair<T, T> array(T[] arr, Comparator<? super T> c) {  
 ...  
 return Pair.of(min, max);  
}Copy

126. Union Find

The Union Find algorithm operates on a *disjoint-set* data structure.

A disjoint-set data structure defines sets of elements separated in certain disjoint subsets that are not overlapping. Graphically, we can represent a disjoint-set with three subsets, as in the following diagram:

Imagen de la pantalla de un celular con la imagen de una caricatura

Descripción generada automáticamente con confianza baja

In the code, a disjoint-set is represented as follows:

* n is the total number of elements (for example, in the preceding diagram, n is 11).
* rank is an array initialized with 0 that is useful to decide how to union two subsets with multiple elements (subsets with lower rank become children of subsets with a higher rank).
* parent is the array that allows us to build an array-based Union Find (initially, parent[0] = 0; parent[1] = 1; ... parent[10] = 10;):

public DisjointSet(int n) {  
  
 this.n = n;  
 rank = new int[n];  
 parent = new int[n];  
  
 initializeDisjointSet();  
}Copy

Mainly, the Union Find algorithms should be capable of the following:

* Merging two subsets into a single subset
* Returning its subset for the given element (this is useful for finding elements that are in the same subset)

In order to store a disjoint-set data structure in memory, we can represent it as an array. Initially, at each index of the array, we store that index (x[i] = i). Each index can be mapped to a piece of meaningful information for us, but this is not mandatory. For example, such an array can be shaped as in the following diagram (initially, we have 11 subsets and each element is a parent of itself):

Imagen que contiene Tabla

Descripción generada automáticamente

Or, if we use numbers, we can represent it in the following diagram:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

In terms of code lines, we have the following:

private void initializeDisjointSet() {  
  
 for (int i = 0; i < n; i++) {  
 parent[i] = i;  
 }  
}Copy

Furthermore, we need to define our subsets via the *union* operation. We can define subsets via a sequence of (*parent*, *child*) pairs. For example, let's define the following three pairs—union(0,1);, union(4, 9);, and union(6, 5);. Each time an element (subset) becomes a child of another element (subset), it will modify its value to reflect the value of its parent, as in the following diagram:

Diagrama

Descripción generada automáticamente con confianza media

This process continues until we define all our subsets. For example, we can add more unions—union(0, 7);, union(4, 3);, union(4, 2);, union(6, 10);, and union(4, 5);. This will result in the following graphical representation:

Diagrama

Descripción generada automáticamente

As a rule of thumb, it is advisable to union smaller subsets to larger subsets and not vice versa. For example, check the moment when we unify the subset that contains **4** with the subset that contains **5**. At that moment, **4** is the parent of the subset and it has three children (**2**, **3**, and **9**), while **5** is next to **10**, the two children of **6**. So, the subset that contains **5** has three nodes (**6**, **5**, **10**), while the subset that contains **4** has four nodes (**4**, **2**, **3**, **9**). So, **4** becomes the parent of **6** and, implicitly, the parent of **5**.

In code lines, this is the job of the rank[] array:

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

Let's now take a look at how to implement the find and union operation.

Implementing the find operation

Finding the subset of the given element is a recursive process that traverses the subset by following the parent elements until the current element is the parent of itself (root element):

public int find(int x) {  
  
 if (parent[x] == x) {  
 return x;  
 } else {  
 return find(parent[x]);  
 }  
}Copy

Implementing the union operation

The *union* operation begins by fetching the root elements of the given subsets. Furthermore, if these two roots are different, they need to rely on their rank to decide which one will become the parent of the other one (the bigger rank becomes a parent). If they have the same rank, then choose one of them and increase its rank by 1:

public void union(int x, int y) {  
  
 int xRoot = find(x);  
 int yRoot = find(y);  
  
 if (xRoot == yRoot) {  
 return;  
 }  
  
 if (rank[xRoot] < rank[yRoot]) {  
 parent[xRoot] = yRoot;  
 } else if (rank[yRoot] < rank[xRoot]) {  
 parent[yRoot] = xRoot;  
 } else {  
 parent[yRoot] = xRoot;  
 rank[xRoot]++;  
 }  
}Copy

OK. Let's now define a disjoint set:

DisjointSet set = new DisjointSet(11);  
set.union(0, 1);  
set.union(4, 9);  
set.union(6, 5);  
set.union(0, 7);  
set.union(4, 3);  
set.union(4, 2);  
set.union(6, 10);  
set.union(4, 5);Copy

And now let's play with it:

// is 4 and 0 friends => false  
System.out.println("Is 4 and 0 friends: "   
 + (set.find(0) == set.find(4)));  
  
// is 4 and 5 friends => true  
System.out.println("Is 4 and 5 friends: "   
 + (set.find(4) == set.find(5)));Copy

This algorithm can be optimized by compressing the paths between elements. For example, check the following diagram:

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

On the left-hand side, in trying to find the parent of **5**, you must pass through **6** until it reaches **4**. Similarly, in trying to find the parent of **10**, you must pass through **6** until it reaches **4**. However, on the right-hand side, we compress the paths of **5** and **10** by linking them directly to **4**. This time, we can find the parent of **5** and **10** without passing through intermediary elements.

Path compression can take place in relation to the find() operation, as follows:

public int find(int x) {  
  
 if (parent[x] != x) {  
 return parent[x] = find(parent[x]);  
 }  
  
 return parent[x];  
}Copy

The code bundled to this book contains both applications, with and without path compression.

127. Fenwick Tree or Binary Indexed Tree

The **Fenwick Tree** (**FT**) or **Binary Indexed Tree** (**BIT**) is an array built to store sums corresponding to another given array. The built array has the same size as the given array, and each position (or node) of the built array stores the sum of some elements of the given array. Since BIT stores partial sums of the given array, it is a very efficient solution for computing the sum of elements of the given array between two given indexes (range sum/queries) by avoiding looping between the indexes and computing the sum.

The BIT can be constructed in linear time or O(n log n). Obviously, we prefer it in linear time, so let's see how we can do this. We begin with the given (original) array that can be (the subscripts represent the index in the array):

3(1), 1(2), 5(3), 8(4), 12(5), 9(6), 7(7), 13(8), 0(9), 3(10), 1(11), 4(12), 9(13), 0(14), 11(15), 5(16)

The idea of building the BIT relies on the **Least Significant Bit** (**LSB**) concept. More precisely, let's assume that we are currently dealing with the element from the index, *a*. Then, the value immediately above us must be at index *b*, where *b = a + LSB(a)*. In order to apply the algorithm, the value from index 0 must be 0; therefore the array that we operate is as follows:

**0(0)**, 3(1), 1(2), 5(3), 8(4), 12(5), 9(6), 7(7), 13(8), 0(9), 3(10), 1(11), 4(12), 9(13), 0(14), 11(15), 5(16)

Now, let's apply a few steps of the algorithm and let's populate the BIT with sums. At index 0 in BIT, we have 0. Furthermore, we use the *b = a + LSB(a)* formula to compute the remaining sums, as follows:

1. a = 1: If a = 1 = 000012, then b = 000012 + 000012 = 1 + 1 = 2 = 000102. We say that 2 is responsible for a (which is 1). Therefore, in BIT, at index 1, we store the value 3, and, at index 2, we store the value sum, 3 + 1 = 4.
2. a = 2: If a = 2 = 000102, then b = 000102 + 000102 = 2 + 2 = 4 = 001002. We say that 4 is responsible for a (which is 2). Therefore, in BIT, at index 4, we store the value sum, 8 + 4 = 12.
3. a = 3: If a = 3 = 000112, then b = 000112 + 000012 = 3 + 1 = 4 = 001002. We say that 4 is responsible for a (which is 3). Therefore, in BIT, at index 4, we store the value sum, 12 + 5 = 17.
4. a = 4. If a = 4 = 001002, then b = 001002 + 001002 = 4 + 4 = 8 = 010002. We say that 8 is responsible for a (which is 4). Therefore, in BIT, at index 8, we store the value sum, 13 + 17 = 30.

The algorithm will continue in the same manner until the BIT is complete. In a graphical representation, our case can be shaped as follows:

Imagen que contiene Diagrama

Descripción generada automáticamente

If a computed point of an index is out of bounds, then simply ignore it.

In code lines, the preceding flow can be shaped as follows (values are the given array):

public class FenwickTree {  
  
 private final int n;  
 private long[] tree;  
 ...  
  
 public FenwickTree(long[] values) {  
  
 values[0] = 0 L;  
 this.n = values.length;  
 tree = values.clone();  
  
 for (int i = 1; i < n; i++) {  
  
 int parent = i + lsb(i);  
 if (parent < n) {  
 tree[parent] += tree[i];  
 }  
 }  
 }  
  
 private static int lsb(int i) {  
  
 return i & -i;  
  
 // or  
 // return Integer.lowestOneBit(i);  
 }  
  
 ...  
}Copy

Now, the BIT is ready and we can perform updates and range queries.

For example, in order to perform range sums, we have to fetch the corresponding ranges and total them up. Consider a few examples on the right-hand side of the following diagram to quickly understand this process:

Imagen que contiene texto, computadora

Descripción generada automáticamente

In terms of code lines, this can be easily shaped as follows:

public long sum(int left, int right) {  
  
 return prefixSum(right) - prefixSum(left - 1);  
}  
  
private long prefixSum(int i) {  
  
 long sum = 0L;  
  
 while (i != 0) {  
 sum += tree[i];  
 i &= ~lsb(i); // or, i -= lsb(i);  
 }  
  
 return sum;  
}Copy

Moreover, we can add a new value:

public void add(int i, long v) {  
  
 while (i < n) {  
 tree[i] += v;  
 i += lsb(i);  
 }  
}Copy

And we can also set a new value to a certain index:

public void set(int i, long v) {  
 add(i, v - sum(i, i));  
}Copy

Having all these features in place, we can create the BIT for our array as follows:

FenwickTree tree = new FenwickTree(new long[] {  
 0, 3, 1, 5, 8, 12, 9, 7, 13, 0, 3, 1, 4, 9, 0, 11, 5  
});Copy

And then we can play with it:

long sum29 = tree.sum(2, 9); // 55  
tree.set(4, 3);  
tree.add(4, 5);Copy

128. Bloom filter

The Bloom filter is a fast and memory-efficient data structure capable of providing a probabilistic answer to the question *Is value X in the given set?*

Commonly, this algorithm is useful when the set is huge and most searching algorithms are facing memory and speed issues.

The speed and memory efficiency of the Bloom filter come from the fact that this data structure relies on an array of bits (for example, java.util.BitSet). Initially, the bits of this array are set to 0 or false.

The array of bits is the first main ingredient of the Bloom filter. The second main ingredient consists of one or more hash functions. Ideally, these are *pairwise independent* and *uniformly distributed* hash functions. Also, it is very important to be extremely fast. Murmur, the fnv series, and HashMix are some of the hash functions that respect these constraints to an acceptable extent for being used by the Bloom filter.

Now, when we add an element to the Bloom filter, we need to hash this element (pass it through each available hash function) and set the bits in the bit array at the index of those hashes to 1 or true.

The following snippet of code should clarify the main idea:

private BitSet bitset; // the array of bits  
private static final Charset CHARSET = StandardCharsets.UTF\_8;  
...  
public void add(T element) {  
  
 add(element.toString().getBytes(CHARSET));  
}  
  
public void add(byte[] bytes) {  
  
 int[] hashes = hash(bytes, numberOfHashFunctions);  
  
 for (int hash: hashes) {  
 bitset.set(Math.abs(hash % bitSetSize), true);  
 }  
  
 numberOfAddedElements++;  
}Copy

Now, when we search for an element, we pass this element through the same hash functions. Furthermore, we check whether the resultant values are marked in the array of bits as 1 or true. If they are not, then the element is not in the set for sure. But if they are, then we know with a certain probability that the element is in the set. This is not 100% certain since another element or combination of elements may have been flipped up those bits. Wrong answers are known as *false positives*.

In terms of code lines, we have the following:

private BitSet bitset; // the array of bits  
private static final Charset CHARSET = StandardCharsets.UTF\_8;  
...  
  
public boolean contains(T element) {  
  
 return contains(element.toString().getBytes(CHARSET));  
}  
  
public boolean contains(byte[] bytes) {  
  
 int[] hashes = hash(bytes, numberOfHashFunctions);  
  
 for (int hash: hashes) {  
 if (!bitset.get(Math.abs(hash % bitSetSize))) {  
  
 return false;  
 }  
 }  
  
 return true;  
}Copy

In a graphical representation, we can represent a Bloom filter with an array of bits of size 11 and three hash functions as follows (we have added two elements):

Diagrama

Descripción generada automáticamente

Obviously, we want to reduce the number of *false positives* as much as possible. While we cannot totally eliminate them, we can still affect their rate by joggling with the size of the bit array, the number of hash functions, and the number of elements in the set.

The following mathematical formulas can be used to shape the optimal Bloom filter:

* Number of items in the filter (can be estimated based on m, k, and p):

n = ceil(m / (-k / log(1 - exp(log(p) / k))));

* Probability of *false positives*, a fraction between 0 and 1, or a number indicating 1-in-p:

p = pow(1 - exp(-k / (m / n)), k);

* Number of bits in the filter (or size in terms of KB, KiB, MB, Mb, GiB, and so on):

m = ceil((n \* log(p)) / log(1 / pow(2, log(2))));

* Number of hash functions (can be estimated based on m and n):

k = round((m / n) \* log(2));

As a rule of thumb, a larger filter will have fewer *false positives* than a smaller one. Moreover, by increasing the number of hash functions, we obtain fewer *false positives*, but we slow down the filter and will fill it up quickly. The performance of the Bloom filter is O(h), where *h* is the number of hash functions.

In the code bundled to this book, there is an implementation of the Bloom filter using hash functions based on SHA-256 and murmur. Since this code is too big to be listed in this book, please consider as a starting point the example from the Main class.

Summary

This chapter covered 30 problems involving arrays, collections, and several data structures. While the problems covering arrays and collections are part of daily work, the problems covering data structures have introduced a few less well-known (but powerful) data structures, such as FT, Union Find, and Trie.

Download the applications from this chapter to see the results and to examine additional details.

Java I/O Paths, Files, Buffers, Scanning, and Formatting

This chapter includes 20 problems that involve Java I/O for files. From manipulating, walking, and watching paths to streaming files and efficient ways for reading/writing text and binary files, we will cover problems that Java developers may face on a day-to-day basis.

With the skills you will have gained from this chapter, you will be able to tackle most of the common problems that involve Java I/O files. The wide range of topics in this chapter will provide a plethora of information about how Java tackles I/O tasks.

Problems

Take a look at the following problems in order to test your Java I/O programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Creating file paths**: Write several examples of creating several kinds of file paths (for example, absolute paths, relative paths, and so on).
2. **Converting file paths**: Write several examples of converting file paths (for example, converting a file path into a string, URI, file, and so on).
3. **Joining file paths**: Write several examples of joining (combining) file paths. Define a fixed path and append other different paths to it (or replace a part of it with other paths).
4. **Constructing a path between two locations**: Write several examples that construct a relative path between two given paths (from one path to another).
5. **Comparing file paths**: Write several examples of comparing the given file paths.
6. **Walking paths**: Write a program that visits all the files within a directory, including subdirectories. Moreover, write a program that searches a file by name, deletes a directory, moves a directory, and copies a directory.
7. **Watching paths**: Write several programs that watch changes that occur on a certain path (for example, create, delete, and modify).
8. **Streaming a file's content**: Write a program that streams the content of the given file.
9. **Searching for files/folders in a file tree**: Write a program that searches for the given files/folders in the given file tree.
10. **Reading/writing text files efficiently**: Write several programs to exemplify different approaches for reading and writing a text file in an efficient manner.
11. **Reading/writing binary files efficiently**: Write several programs to exemplify different approaches for reading and writing a binary file in an efficient manner.
12. **Searching in big files**: Write a program that efficiently searches the given string in a big file.
13. **Reading a JSON/CSV file as an object**: Write a program that reads the given JSON/CSV file as an object (POJO).
14. **Working with temporary files/folders**: Write several programs for working with temporary files/folders.
15. **Filtering files**: Write several user-defined filters for files.
16. **Discovering mismatches between two files**: Write a program that discovers the mismatches between two files at the byte level.
17. **Circular byte buffer**: Write a program that represents an implementation of a circular byte buffer.
18. **Tokenizing files**: Write several snippets of code to exemplify different techniques of tokenizing a file content.
19. **Writing formatted output directly to a file**: Write a program that formats the given numbers (integers and doubles) and outputs them to a file.
20. **Working with** Scanner: Write several snippets of code to reveal Scanner capabilities.

Solutions

The following sections describe the solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations that are shown here only include the most interesting and important details that are needed to solve the problems. You can download the example solutions to view additional details and experiment with the programs from <https://github.com/PacktPublishing/Java-Coding-Problems>.

129. Creating file paths

Starting with JDK 7, we can create a file path via the NIO.2 API. More precisely, a file path can be easily defined via the Path and Paths APIs.

The Path class is a programmatic representation of a path in a filesystem. The path string contains the following information:

* The filename
* The directories list
* The OS-dependent file delimiter (for example, a forward slash / on Solaris and Linux and a backslash \ on Microsoft Windows)
* Other allowed characters, for example, the . (current directory) and .. (parent directory) notations

The Path class works with files in different filesystems (FileSystem) that can use different storage places (FileStore is the underlying storage).

A common solution for defining a Path is to call one of the get() methods of the Paths helper class. Another solution relies on the FileSystems.getDefault().getPath() method.

A Path resides in a filesystem—*a filesystem stores and organizes files or some form of media, generally on one or more hard drives, in such a way that they can be easily retrieved*. The filesystem can be obtained through the final class of java.nio.file.FileSystems, which is used to get an instance of java.nio.file.FileSystem. The default FileSystem of the JVM (commonly known as the default filesystem of the operating system) can be obtained via the FileSystems().getDefault() method. Once we know the filesystem and the location of a file (or directory/folder), we can create a Path object for it.

Another approach consists of creating a Path from a **Uniform Resource Identifier** (**URI**). Java wraps a URI via the URI class; then, we can obtain a URI from a String via the URI.create(String uri) method. Furthermore, the Paths class provides a get() method that takes a URI object as an argument and returns the corresponding Path.

Starting with JDK 11, we can create a Path via two of() methods. One of them converts a URI into a Path, while the other one converts a path-string, or a sequence of strings, joined as a path-string.

In the upcoming sections, we'll take a look at the various ways we can create paths.

Creating a path relative to the file store root

A path that's relative to the current file store root (for example, C:/) must start with the file delimiter. In the following examples, if the current file store root is C, then the absolute path is C:\learning\packt\JavaModernChallenge.pdf:

Path path = Paths.get("/learning/packt/JavaModernChallenge.pdf");  
Path path = Paths.get("/learning", "packt/JavaModernChallenge.pdf");  
  
Path path = Path.of("/learning/packt/JavaModernChallenge.pdf");  
Path path = Path.of("/learning", "packt/JavaModernChallenge.pdf");  
  
Path path = FileSystems.getDefault()  
 .getPath("/learning/packt", "JavaModernChallenge.pdf");  
Path path = FileSystems.getDefault()  
 .getPath("/learning/packt/JavaModernChallenge.pdf");  
  
Path path = Paths.get(  
 URI.create("file:///learning/packt/JavaModernChallenge.pdf"));  
Path path = Path.of(  
 URI.create("file:///learning/packt/JavaModernChallenge.pdf"));Copy

Creating a path relative to the current folder

When we create a path that's relative to the current working folder, the path should *not* start with the file delimiter. If the current folder is named books and is under the C root, then the absolute path that's returned by the following snippet of code will be C:\books\learning\packt\JavaModernChallenge.pdf:

Path path = Paths.get("learning/packt/JavaModernChallenge.pdf");  
Path path = Paths.get("learning", "packt/JavaModernChallenge.pdf");  
  
Path path = Path.of("learning/packt/JavaModernChallenge.pdf");  
Path path = Path.of("learning", "packt/JavaModernChallenge.pdf");  
  
Path path = FileSystems.getDefault()  
 .getPath("learning/packt", "JavaModernChallenge.pdf");  
Path path = FileSystems.getDefault()  
 .getPath("learning/packt/JavaModernChallenge.pdf");Copy

Creating an absolute path

Creating an absolute path can be accomplished by explicitly specifying the root directory and all other subdirectories that contain the file or folder, as shown in the following examples (C:\learning\packt\JavaModernChallenge.pdf):

Path path = Paths.get("C:/learning/packt", "JavaModernChallenge.pdf");  
Path path = Paths.get(  
 "C:", "learning/packt", "JavaModernChallenge.pdf");  
Path path = Paths.get(  
 "C:", "learning", "packt", "JavaModernChallenge.pdf");  
Path path = Paths.get("C:/learning/packt/JavaModernChallenge.pdf");  
Path path = Paths.get(  
 System.getProperty("user.home"), "downloads", "chess.exe");  
  
Path path = Path.of(  
 "C:", "learning/packt", "JavaModernChallenge.pdf");  
Path path = Path.of(  
 System.getProperty("user.home"), "downloads", "chess.exe");  
  
Path path = Paths.get(URI.create(  
 "file:///C:/learning/packt/JavaModernChallenge.pdf"));  
Path path = Path.of(URI.create(  
 "file:///C:/learning/packt/JavaModernChallenge.pdf"));Copy

Creating a path using shortcuts

We understand shortcuts to be the . (current directory) and .. (parent directory) notations. This kind of path can be normalized via the normalize() method. This method eliminates redundancies such as . and directory/..:

Path path = Paths.get(  
 "C:/learning/packt/chapters/../JavaModernChallenge.pdf")  
 .normalize();  
Path path = Paths.get(  
 "C:/learning/./packt/chapters/../JavaModernChallenge.pdf")  
 .normalize();  
  
Path path = FileSystems.getDefault()  
 .getPath("/learning/./packt", "JavaModernChallenge.pdf")  
 .normalize();  
  
Path path = Path.of(  
 "C:/learning/packt/chapters/../JavaModernChallenge.pdf")  
 .normalize();  
Path path = Path.of(  
 "C:/learning/./packt/chapters/../JavaModernChallenge.pdf")  
 .normalize();Copy

Without normalization, the redundant parts of the path will not be removed.

For creating paths that are 100% compatible with the current operating system, we can rely on FileSystems.getDefault().getPath(), or a combination of File.separator (system-dependent default name separator character) and File.listRoots() (the available filesystem roots). For relative paths, we can rely on the following examples:

private static final String FILE\_SEPARATOR = File.separator;Copy

Alternatively, we can rely on getSeparator():

private static final String FILE\_SEPARATOR  
 = FileSystems.getDefault().getSeparator();  
  
// relative to current working folder  
Path path = Paths.get("learning",  
 "packt", "JavaModernChallenge.pdf");  
Path path = Path.of("learning",  
 "packt", "JavaModernChallenge.pdf");  
Path path = Paths.get(String.join(FILE\_SEPARATOR, "learning",  
 "packt", "JavaModernChallenge.pdf"));  
Path path = Path.of(String.join(FILE\_SEPARATOR, "learning",  
 "packt", "JavaModernChallenge.pdf"));  
  
// relative to the file store root  
Path path = Paths.get(FILE\_SEPARATOR + "learning",  
 "packt", "JavaModernChallenge.pdf");  
Path path = Path.of(FILE\_SEPARATOR + "learning",  
 "packt", "JavaModernChallenge.pdf");Copy

We can also do the same for absolute paths:

Path path = Paths.get(File.listRoots()[0] + "learning",  
 "packt", "JavaModernChallenge.pdf");  
Path path = Path.of(File.listRoots()[0] + "learning",  
 "packt", "JavaModernChallenge.pdf");Copy

The list of root directories can be obtained via FileSystems as well:

FileSystems.getDefault().getRootDirectories()Copy

130. Converting file paths

Converting a file path into a String, URI, File, and so on is a common task that can occur in a wide range of applications. Let's consider the following file path:

Path path = Paths.get("/learning/packt", "JavaModernChallenge.pdf");Copy

Now, based on JDK 7 and the NIO.2 API, let's see how we can convert a Path into a String, a URI, an absolute path, a *real* path, and a file:

* Converting a Path into a String is as simple as calling (explicitly or automatically) the Path.toString() method. Notice that if the path was obtained via the FileSystem.getPath() method, then the path-string returned by toString() may differ from the initial String that was used to create the path:

// \learning\packt\JavaModernChallenge.pdf  
String pathToString = path.toString();Copy

* Converting a Path into a URI (browser format) can be accomplished via the Path.toURI() method. The returned URI wraps a path-string that can be used in the address bar of a web browser:

// file:///D:/learning/packt/JavaModernChallenge.pdf  
URI pathToURI = path.toUri();Copy

Let's say that we want to extract the filename present in a URI/URL as Path (this is a common scenario to encounter). In such cases, we can rely on the following snippets of code:

// JavaModernChallenge.pdf  
URI uri = URI.create(  
 "https://www.learning.com/packt/JavaModernChallenge.pdf");  
Path URIToPath = Paths.get(uri.getPath()).getFileName();  
  
// JavaModernChallenge.pdf  
URL url = new URL(  
 "https://www.learning.com/packt/JavaModernChallenge.pdf");  
Path URLToPath = Paths.get(url.getPath()).getFileName();Copy

Conversion of paths can be done as follows:

* Converting a relative Path into an absolute Path can be done via the Path.toAbsolutePath() method. If the Path is already absolute, then the same result will be returned:

// D:\learning\packt\JavaModernChallenge.pdf  
Path pathToAbsolutePath = path.toAbsolutePath();Copy

* Converting a Path into a *real* Path can be accomplished via the Path.toRealPath() method and its result is dependent on the implementation. If the file that's being pointed to doesn't exist, then this method will throw an IOException. But, as a rule of thumb, the result of calling this method is an absolute path without redundant elements (normalized). This method gets an argument that indicates how *symbolic links* should be treated. By default, if the filesystem supports *symbolic links*, then this method will try to resolve them. If you wish to ignore the *symbolic links*, simply pass the LinkOption.NOFOLLOW\_LINKS constant to the method. Moreover, the path name elements will represent the actual name of the directories and the file.

For example, let's consider the following Path and the result of calling this method (notice that we have intentionally added several redundant elements and capitalized the PACKT folder):

Path path = Paths.get(  
 "/learning/books/../PACKT/./", "JavaModernChallenge.pdf");  
  
// D:\learning\packt\JavaModernChallenge.pdf  
Path realPath = path.toRealPath(LinkOption.NOFOLLOW\_LINKS);Copy

* Converting a Path into a file can be done via the Path.toFile() method. For converting a file into a Path, we can rely on the File.toPath() method:

File pathToFile = path.toFile();  
Path fileToPath = pathToFile.toPath();Copy

131. Joining file paths

Joining (or combining) file paths means defining a fixed root path and appending to it a partial path or replacing a part of it (for example, a filename needs to be replaced with another filename). Basically, this is a handy technique when we want to create new paths that share a common fixed part.

This can be accomplished via NIO.2 and the Path.resolve() and Path.resolveSibling() methods.

Let's consider the following fixed root path:

Path base = Paths.get("D:/learning/packt");Copy

Let's also assume that we want to obtain the Path for two different books:

// D:\learning\packt\JBossTools3.pdf  
Path path = base.resolve("JBossTools3.pdf");  
  
// D:\learning\packt\MasteringJSF22.pdf  
Path path = base.resolve("MasteringJSF22.pdf");Copy

We can use this feature to loop a set of files; for example, let's loop a String[] of books:

Path basePath = Paths.get("D:/learning/packt");  
String[] books = {  
 "Book1.pdf", "Book2.pdf", "Book3.pdf"  
};  
  
for (String book: books) {  
 Path nextBook = basePath.resolve(book);  
 System.out.println(nextBook);  
}Copy

Sometimes, the fixed root path contains the filename as well:

Path base = Paths.get("D:/learning/packt/JavaModernChallenge.pdf");Copy

This time, we can replace the name of the file (JavaModernChallenge.pdf) with another name via the resolveSibling() method. This method resolves the given path against this path's parent path, as shown in the following example:

// D:\learning\packt\MasteringJSF22.pdf  
Path path = base.resolveSibling("MasteringJSF22.pdf");Copy

If we bring the Path.getParent() method into the discussion and we chain the resolve() and resolveSibling() methods, then we can create more complex paths, as shown in the following example:

// D:\learning\publisher\MyBook.pdf  
Path path = base.getParent().resolveSibling("publisher")  
 .resolve("MyBook.pdf");Copy

The resolve()/resolveSibling() method comes in two flavors – resolve​(String other) / resolveSibling​(String other) and resolve​(Path other) / resolveSibling​(Path other), respectively.

132. Constructing a path between two locations

Constructing a relative path between two locations is a job for the Path.relativize() method.

Basically, the resulted relative path (returned by Path.relativize()) starts from a path and ends on another path. This is a powerful feature that allows us to navigate between different locations using relative paths that are resolved against the previous paths.

Let's consider the following two paths:

Path path1 = Paths.get("JBossTools3.pdf");  
Path path2 = Paths.get("JavaModernChallenge.pdf");Copy

Notice that JBossTools3.pdf and JavaModernChallenge.pdf are siblings. This means that we can navigate from one to another by going up one level and then down one level. This navigation case is revealed by the following examples as well:

// ..\JavaModernChallenge.pdf  
Path path1ToPath2 = path1.relativize(path2);  
  
// ..\JBossTools3.pdf  
Path path2ToPath1 = path2.relativize(path1);Copy

Another common case involves a common root element:

Path path3 = Paths.get("/learning/packt/2003/JBossTools3.pdf");  
Path path4 = Paths.get("/learning/packt/2019");Copy

So, path3 and path4 share the same common root element, /learning. For navigating from path3 to path4, we need to go up two levels and down one level. In addition, for navigating from path4 to path3, we need to go up one level and down two levels. Check out the following code:

// ..\..\2019  
Path path3ToPath4 = path3.relativize(path4);  
  
// ..\2003\JBossTools3.pdf  
Path path4ToPath3 = path4.relativize(path3);Copy

Both paths must include a root element. Accomplishing this requirement does not guarantee success because the construction of the relative path is implementation-dependent.

133. Comparing file paths

Depending on how we perceive the equality between two file paths, there are several solutions to this problem. Mainly, Path equality can be verified in different ways for different goals.

Let's assume that we have the following three paths (consider reproducing path3 on your computer):

Path path1 = Paths.get("/learning/packt/JavaModernChallenge.pdf");  
Path path2 = Paths.get("/LEARNING/PACKT/JavaModernChallenge.pdf");  
Path path3 = Paths.get("D:/learning/packt/JavaModernChallenge.pdf");Copy

In the following sections, we'll take a look at the different methods that are used to compare file paths.

Path.equals()

Is path1 equal to path2? Or, is path2 equal to path3? Well, if we perform these tests via Path.equals(), then a possible result will reveal that path1 is equal to path2, but path2 is not equal to path3:

boolean path1EqualsPath2 = path1.equals(path2); // true  
boolean path2EqualsPath3 = path2.equals(path3); // falseCopy

The Path.equals() method follows the Object.equals() specification. While this method doesn't access the filesystem, equality depends on the filesystem implementation. For example, some filesystem implementations may compare paths in a case-sensitive manner, while others may ignore case.

Paths representing the same file/folder

However, this probably isn't the kind of comparison that we want. It is more meaningful to say that two paths are equal if they are the same file or folder. This can be accomplished via the Files.isSameFile() method. This method acts in two steps:

1. First, it calls Path.equals(), and, if this method returns true, then the paths are equal and need no further action.
2. Second, if Path.equals() returns false, then it checks if both paths represent the same file/folder (depending on the implementation, this action may need to open/access both files, so the files must exist in order to avoid an IOException).

//true  
boolean path1IsSameFilePath2 = Files.isSameFile(path1, path2);  
//true  
boolean path1IsSameFilePath3 = Files.isSameFile(path1, path3);  
//true  
boolean path2IsSameFilePath3 = Files.isSameFile(path2, path3);Copy

Lexicographical comparison

If all we want is a lexicographical comparison of the paths, then we can rely on the Path.compareTo() method (this can be useful for sorting).

This method returns the following information:

* 0 if the paths are equal
* A value less than zero if the first path is lexicographically less than the argument path
* A value greater than zero if the first path is lexicographically greater than the argument path:

int path1compareToPath2 = path1.compareTo(path2); // 0  
int path1compareToPath3 = path1.compareTo(path3); // 24  
int path2compareToPath3 = path2.compareTo(path3); // 24Copy

Note that you may obtain different values than in the preceding example. Furthermore, in your business logic, it is important to rely on their meaning and not on their value (for example, say if(path1compareToPath3 > 0) { ... } and avoid if(path1compareToPath3 == 24) { ... }).

Partial comparing

Partial comparing is achievable via the Path.startsWith() and Path.endsWith() methods. Using these methods, we can test whether the current path starts/ends with the given path:

boolean sw = path1.startsWith("/learning/packt"); // true  
boolean ew = path1.endsWith("JavaModernChallenge.pdf"); // trueCopy

134. Walking paths

There are different solutions for walking (or visiting) paths, and one of them is provided by the NIO.2 API via the FileVisitor interface.

This interface exposes a set of methods that represent checkpoints in the recursive process of visiting the given path. By overriding these checkpoints, we are allowed to interfere in this process. We can process the currently visited file/folder and decide what should happen further via the FileVisitResult enumeration, which contains the following constants:

* CONTINUE: The traversal process should continue (visit next file, folder, skip a failure, and so on)
* SKIP\_SIBLINGS: The traversal process should continue without visiting the siblings of the current file/folder
* SKIP\_SUBTREE: The traversal process should continue without visiting the entries in the current folder
* TERMINATE: The traversal should brutally terminate

The methods that are exposed by FileVisitor are as follows:

* FileVisitResult visitFile​(T file, BasicFileAttributes attrs) throws IOException: Automatically called for each visited file/folder
* FileVisitResult preVisitDirectory​(T dir, BasicFileAttributes attrs) throws IOException: Automatically called for a folder before visiting its content
* FileVisitResult postVisitDirectory​(T dir, IOException exc) throws IOException: Automatically called after the content in the directory (including descendants) is visited or, during the iteration of the folder, an I/O error occurred or the visit was programmatically aborted
* FileVisitResult visitFileFailed​(T file, IOException exc) throws IOException: Automatically called when the file cannot be visited (accessed) for different reasons (for example, the file's attributes cannot be read or a folder cannot be opened)

Ok; so far, so good! Let's continue with several practical examples.

Trivial traversal of a folder

Implementing the FileVisitor interface requires that we override its four methods. However, NIO.2 comes with a built-in simple implementation of this interface called SimpleFileVisitor. For simple cases, extending this class is more convenient than implementing FileVisitor since it allows us to override only the necessary methods.

For example, let's assume that we store our e-courses in the subfolders of the D:/learning folder, and we want to visit each of these subfolders via the FileVisitor API. If something goes wrong during the iteration of a subfolder, we will simply throw the reported exception.

In order to shape this behavior, we need to override the postVisitDirectory() method, as follows:

class PathVisitor extends SimpleFileVisitor<Path> {  
  
 @Override  
 public FileVisitResult postVisitDirectory(  
 Path dir, IOException ioe) throws IOException {  
  
 if (ioe != null) {  
 throw ioe;  
 }  
  
 System.out.println("Visited directory: " + dir);  
  
 return FileVisitResult.CONTINUE;  
 }  
}Copy

In order to use the PathVisitor class, we just need to set up the path and call one of the Files.walkFileTree() methods, as follows (the flavor of walkFileTree() that's used here gets the starting file/folder and the corresponding FileVisitor):

Path path = Paths.get("D:/learning");  
PathVisitor visitor = new PathVisitor();  
  
Files.walkFileTree(path, visitor);Copy

By using the preceding code, we will receive the following output:

Visited directory: D:\learning\books\ajax  
Visited directory: D:\learning\books\angular  
...Copy

Searching for a file by name

Searching a certain file on a computer is a common task. Typically, we rely on tools that are provided by the operating system or additional tools, but if we want to accomplish this programmatically (for example, we may want to write a file search tool with special features), then FileVisitor can help us achieve this in a pretty straightforward way. The stub of this application is listed as follows:

public class SearchFileVisitor implements FileVisitor {  
  
 private final Path fileNameToSearch;  
 private boolean fileFound;  
 ...  
  
 private boolean search(Path file) throws IOException {  
  
 Path fileName = file.getFileName();  
  
 if (fileNameToSearch.equals(fileName)) {  
 System.out.println("Searched file was found: " +  
 fileNameToSearch + " in " + file.toRealPath().toString());  
  
 return true;  
 }  
  
 return false;  
 }  
}Copy

Let's take a look at the main checkpoints and the implementation of searching a file by name:

* visitFile() is our main checkpoint. Once we have control, we can query the currently visited file for its name, extension, attributes, and so on. This information is needed in order to draw a comparison with the same information on the searched file. For example, we compare the names, and at first match, we TERMINATE the search. But if we search for more such files (if we know that there is more than one), then we can return CONTINUE:

@Override  
public FileVisitResult visitFile(  
 Object file, BasicFileAttributes attrs) throws IOException {  
  
 fileFound = search((Path) file);  
  
 if (!fileFound) {  
 return FileVisitResult.CONTINUE;  
 } else {  
 return FileVisitResult.TERMINATE;  
 }  
}Copy

The visitFile() method cannot be used for finding folders. Use the preVisitDirectory() or postVisitDirectory() methods instead.

* visitFileFailed() is the second important checkpoint. When this method is invoked, we know that something went wrong while visiting the current file. We prefer to ignore any such issues and CONTINUE the search. It's pointless to stop the search process:

@Override  
public FileVisitResult visitFileFailed(  
 Object file, IOException ioe) throws IOException {  
 return FileVisitResult.CONTINUE;  
}Copy

The preVisitDirectory() and postVisitDirectory() methods don't carry any important tasks, so we can skip them for brevity.

In order to start the search, we rely on another flavor of the Files.walkFileTree() method. This time, we specify the start point of the search (for example, all roots), the options that were used during searching (for example, follow *symbolic links*), the maximum number of directory levels to visit (for example, Integer.MAX\_VALUE), and the FileVisitor (for example, SearchFileVisitor):

Path searchFile = Paths.get("JavaModernChallenge.pdf");  
  
SearchFileVisitor searchFileVisitor   
 = new SearchFileVisitor(searchFile);  
  
EnumSet opts = EnumSet.of(FileVisitOption.FOLLOW\_LINKS);  
Iterable<Path> roots = FileSystems.getDefault().getRootDirectories();  
  
for (Path root: roots) {  
 if (!searchFileVisitor.isFileFound()) {  
 Files.walkFileTree(root, opts,  
 Integer.MAX\_VALUE, searchFileVisitor);  
 }  
}Copy

If you take a look at the code that's bundled with this book, the preceding search traverses all the roots (directories) of your computer in a recursive approach. The preceding example can be easily adapted for searching by extension, by a pattern, or to look inside files from some text.

Deleting a folder

Before attempting to delete a folder, we must delete all the files from it. This statement is very important since it doesn't allow us to simply call the delete()/deleteIfExists() methods for a folder that contains files. An elegant solution to this problem relies on a FileVisitor implementation that starts from the following stub:

public class DeleteFileVisitor implements FileVisitor {  
 ...  
 private static boolean delete(Path file) throws IOException {  
  
 return Files.deleteIfExists(file);  
 }  
}Copy

Let's take a look at the main checkpoints and the implementation of deleting a folder:

* visitFile() is the perfect place for deleting each file from the given folder or subfolder (if a file cannot be deleted, then we simply pass it to the next file, but feel free to adapt the code to suit your needs):

@Override  
public FileVisitResult visitFile(  
 Object file, BasicFileAttributes attrs) throws IOException {  
  
 delete((Path) file);  
  
 return FileVisitResult.CONTINUE;  
}Copy

* A folder can be deleted only if it is empty, and so postVisitDirectory() is the perfect place to do this (we ignore any potential IOException, but feel free to adapt the code to suit your needs (for example, log the names of the folders that couldn't be deleted or throw an exception to stop the process)):

@Override  
public FileVisitResult postVisitDirectory(  
 Object dir, IOException ioe) throws IOException {  
  
 delete((Path) dir);  
  
 return FileVisitResult.CONTINUE;  
}Copy

In visitFileFailed() and preVisitDirectory(), we simply return CONTINUE.

For deleting the folder, in D:/learning, we can call DeleteFileVisitor, as follows:

Path directory = Paths.get("D:/learning");  
DeleteFileVisitor deleteFileVisitor = new DeleteFileVisitor();  
EnumSet opts = EnumSet.of(FileVisitOption.FOLLOW\_LINKS);  
  
Files.walkFileTree(directory, opts,   
 Integer.MAX\_VALUE, deleteFileVisitor);Copy

By combining SearchFileVisitor and DeleteFileVisitor, we can obtain a search-delete application.

Copying a folder

In order to copy a file, we can rely on the Path copy​(Path source, Path target, CopyOption options) throws IOException method. This method copies a file to the target file with the options parameter specifying how the copy is performed.

By combining the copy() method with a custom FileVisitor, we can copy an entire folder (including all its content). The stub code of this custom FileVisitor is listed as follows:

public class CopyFileVisitor implements FileVisitor {  
  
 private final Path copyFrom;  
 private final Path copyTo;  
 ...  
  
 private static void copySubTree(  
 Path copyFrom, Path copyTo) throws IOException {  
  
 Files.copy(copyFrom, copyTo,   
 REPLACE\_EXISTING, COPY\_ATTRIBUTES);  
 }  
}Copy

Let's take a look at the main checkpoints and the implementation of copying a folder (note that we will act indulgently by copying anything that we can and avoid throwing exceptions, but feel free to adapt the code to suit your needs):

* Before copying any files from a source folder, we need to copy the source folder itself. Copying a source folder (empty or not) will result in an empty target folder. This is the perfect task to accomplish in the preVisitDirectory() method:

@Override  
public FileVisitResult preVisitDirectory(  
 Object dir, BasicFileAttributes attrs) throws IOException {  
  
 Path newDir = copyTo.resolve(  
 copyFrom.relativize((Path) dir));  
  
 try {  
 Files.copy((Path) dir, newDir,   
 REPLACE\_EXISTING, COPY\_ATTRIBUTES);  
 } catch (IOException e) {  
 System.err.println("Unable to create "  
 + newDir + " [" + e + "]");  
  
 return FileVisitResult.SKIP\_SUBTREE;  
 }  
  
 return FileVisitResult.CONTINUE;  
}Copy

* The visitFile() method is the perfect place to copy each file:

@Override  
public FileVisitResult visitFile(  
 Object file, BasicFileAttributes attrs) throws IOException {  
  
 try {  
 copySubTree((Path) file, copyTo.resolve(  
 copyFrom.relativize((Path) file)));  
 } catch (IOException e) {  
 System.err.println("Unable to copy "   
 + copyFrom + " [" + e + "]");  
 }  
  
 return FileVisitResult.CONTINUE;  
}Copy

* Optionally, we can preserve the attributes of the source directory. This can be accomplished only after the files have been copied into the postVisitDirectory() method (for example, let's preserve the last modified time):

@Override  
public FileVisitResult postVisitDirectory(  
 Object dir, IOException ioe) throws IOException {  
  
 Path newDir = copyTo.resolve(  
 copyFrom.relativize((Path) dir));  
  
 try {  
 FileTime time = Files.getLastModifiedTime((Path) dir);  
 Files.setLastModifiedTime(newDir, time);  
 } catch (IOException e) {  
 System.err.println("Unable to preserve   
 the time attribute to: " + newDir + " [" + e + "]");  
 }  
  
 return FileVisitResult.CONTINUE;  
}Copy

* If a file cannot be visited, then visitFileFailed() will be invoked. This is a good moment to detect *circular links* and report them. By following links (FOLLOW\_LINKS), we can encounter cases where the file tree has a *circular link* to a parent folder. These cases are reported via FileSystemLoopException exceptions in visitFileFailed():

@Override  
public FileVisitResult visitFileFailed(  
 Object file, IOException ioe) throws IOException {  
  
 if (ioe instanceof FileSystemLoopException) {  
 System.err.println("Cycle was detected: " + (Path) file);  
 } else {  
 System.err.println("Error occured, unable to copy:"  
 + (Path) file + " [" + ioe + "]");  
 }  
  
 return FileVisitResult.CONTINUE;  
}Copy

Let's copy the D:/learning/packt folder to D:/e-courses:

Path copyFrom = Paths.get("D:/learning/packt");  
Path copyTo = Paths.get("D:/e-courses");  
  
CopyFileVisitor copyFileVisitor   
 = new CopyFileVisitor(copyFrom, copyTo);  
  
EnumSet opts = EnumSet.of(FileVisitOption.FOLLOW\_LINKS);  
  
Files.walkFileTree(copyFrom, opts, Integer.MAX\_VALUE, copyFileVisitor);Copy

By combining CopyFileVisitor and DeleteFileVisitor, we can easily shape an application for moving folders. In the code bundled with this book, there is a complete example of moving folders as well. Based on the expertise we've accumulated so far, the code should be pretty accessible without further details.  
  
Pay attention when logging information regarding files (for example, as in the case of handling exceptions) since files (for example, their names, paths, and attributes) may contain sensitive information that can be exploited in a malicious fashion.

JDK 8, Files.walk()

Starting with JDK 8, the Files class has been enriched with two walk() methods. These methods return a Stream that is lazily populated with Path. It does this by walking the file tree that's rooted at a given starting file using the given maximum depth and options:

public static Stream<Path> walk​(  
 Path start, FileVisitOption...options)   
 throws IOException  
  
public static Stream<Path> walk​(  
 Path start, int maxDepth, FileVisitOption...options)   
 throws IOExceptionCopy

For example, let's display all the paths from D:/learning that start with D:/learning/books/cdi:

Path directory = Paths.get("D:/learning");  
  
Stream<Path> streamOfPath = Files.walk(  
 directory, FileVisitOption.FOLLOW\_LINKS);  
  
streamOfPath.filter(e -> e.startsWith("D:/learning/books/cdi"))  
 .forEach(System.out::println);Copy

Now, let's compute the size in bytes for a folder (for example, D:/learning):

long folderSize = Files.walk(directory)  
 .filter(f -> f.toFile().isFile())  
 .mapToLong(f -> f.toFile().length())  
 .sum();Copy

This method is *weakly consistent*. It doesn't freeze the file tree during the iteration process. The potential updates to the file tree may or may not be reflected.

135. Watching paths

Watching paths for changes is just one of the thread-safe goals that can be accomplished via the JDK 7 NIO.2, low-level WatchService API.

In a nutshell, a path can be watched for changes by following two major steps:

1. Register a folder (or folders) to be watched for different kinds of event types.
2. When a registered event type is detected by WatchService, it is handled in a separate thread, so the watch service is not blocked.

At the API level, the starting point is the WatchService interface. This interface comes in different flavors for different file/operating systems.

This interface works hand-in-hand with two main classes. Together, they provide a convenient approach that you can implement to add watching capabilities to a certain context (for example, to the filesystem):

* Watchable: Any object that implements this interface is a *watchable object*, and so it can be watched for changes (for example, Path)
* StandardWatchEventKinds: This class defines the standard *event types* (these are the event types that we can register for notifications:
  + ENTRY\_CREATE: Directory entry created
  + ENTRY\_DELETE: Directory entry deleted
  + ENTRY\_MODIFY: Directory entry modified; what is considered as a modification is somewhat platform-specific, but actually modifying the content of a file should always trigger this event type
  + OVERFLOW: A special event to indicate that events may have been lost or discarded

WatchService is known as the *watcher*, and we say that the *watcher* watches *watchables*. In the following examples, WatchService will be created through the FileSystem class and will watch the registered Path.

Watching a folder for changes

Let's start with a stub method that gets the Path of the folder that should be monitored for changes as an argument:

public void watchFolder(Path path)   
 throws IOException, InterruptedException {  
 ...  
}Copy

WatchService will notify us when any of the ENTRY\_CREATE, ENTRY\_DELETE, and ENTRY\_MODIFY event types occur on the given folder. For this, we need to follow several steps:

1. Create WatchService so that we can monitor the filesystem—this is accomplished via FileSystem.newWatchService(), as follows:

WatchService watchService   
 = FileSystems.getDefault().newWatchService();Copy

1. Register the event types that should be notified—this is accomplished via Watchable.register():

path.register(watchService, StandardWatchEventKinds.ENTRY\_CREATE,  
 StandardWatchEventKinds.ENTRY\_MODIFY,  
 StandardWatchEventKinds.ENTRY\_DELETE);Copy

For each *watchable object*, we receive a registration token as a WatchKey instance (*watch key*). We receive this *watch key* at registration time, but WatchService returns the relevant WatchKey every time an event is triggered.

1. Now, we need to wait for incoming events. This is accomplished in an infinite loop (when an event occurs, the *watcher* is responsible for queuing the corresponding *watch key* for later retrieval and changing its status to *signaled*):

while (true) {  
 // process the incoming event types  
}Copy

1. Now, we need to retrieve a *watch key* – there are at least three methods dedicated to retrieving a *watch key*:
   * poll(): Returns the next key from the queue and removes it (alternatively, it will return null if no key is present).
   * poll​(long timeout, TimeUnit unit): Returns the next key from the queue and removes it; if no key is present, then it waits for the specified timeout and tries again. If a key still isn't available, then it returns null.
   * take(): Returns the next key from the queue and removes it; if no key is present, then it will wait until a key is queued or the infinite loop is stopped:

WatchKey key = watchService.take();Copy

1. Next, we need to retrieve the pending events of a *watch key*. A *watch key* in *signaled* status has at least one pending event; we can retrieve and remove all the events of a certain *watch key* via the WatchKey.pollEvents() method (each event is represented by a WatchEvent instance):

for (WatchEvent<?> watchEvent : key.pollEvents()) {  
 ...  
}Copy

1. Then, we retrieve information about the *event type*. For each event, we can obtain different information (for example, the event type, number of occurrences, and context-specific information (for example, the filename that caused the event), which is useful for processing the event)):

Kind<?> kind = watchEvent.kind();  
WatchEvent<Path> watchEventPath = (WatchEvent<Path>) watchEvent;  
Path filename = watchEventPath.context();Copy

1. Next, we reset the *watch key*. A *watch key* has a status that can be either *ready* (initial status at creation), *signaled*, or *invalid.* Once *signaled*, a *watch key* remains like this until we call the reset() method, which attempts to put it back in the *ready* status to accept the event's state. If the transition from *signaled* to *ready* (resume waiting events) was successful, then the reset() method returns true; otherwise, it returns false, which means that the *watch key* may be *invalid*. A *watch key* can be in an *invalid* state if it is no longer active (inactivity can be caused by explicitly calling the close() method of the *watch key*, closing the watcher, the directory was deleted, and so on):

boolean valid = key.reset();  
  
if (!valid) {  
 break;  
}Copy

When there is a single *watch key* in an *invalid* state, then there is no reason to stay in the infinite loop. Simply call break to jump out of the loop.

1. Finally, we close the *watcher*. This can be accomplished by explicitly calling the close() method of WatchService or relying on *try-with-resources*, as follows:

try (WatchService watchService  
 = FileSystems.getDefault().newWatchService()) {  
 ...  
}Copy

The code that's bundled with this book glues all these snippets of code into a single class named FolderWatcher. The result will be a *watcher* that's capable of reporting the create, delete, and modify events that occurred on the specified path.

In order to watch the path, that is, D:/learning/packt, we just call the watchFolder() method:

Path path = Paths.get("D:/learning/packt");  
  
FolderWatcher watcher = new FolderWatcher();  
watcher.watchFolder(path);Copy

Running the application will display the following message:

Watching: D:\learning\packtCopy

Now, we can create, delete, or modify a file directly under this folder and check the notifications. For example, if we simply copy-paste a file called resources.txt, then the output will be as follows:

ENTRY\_CREATE -> resources.txt  
ENTRY\_MODIFY -> resources.txtCopy

In the end, don't forget to stop the application, since it will run indefinitely (in theory).

Starting with this application, the source code bundled with this book comes with two more applications. One of them is a simulation of a video capture system, while the other is a simulation of a printer tray watcher. By relying on the knowledge that we've accumulated during this section, it should be pretty straightforward to understand these two applications without further details.

136. Streaming a file's content

Streaming a file's content is a problem that can be solved via JDK 8 using the Files.lines() and BufferedReader.lines() methods.

Stream<String> Files.lines​(Path path, Charset cs) reads all the lines from a file as a Stream. This happens lazily, as the stream is consumed. During the execution of the Terminal stream operation, the file's content should not be modified; otherwise, the result is undefined.

Let's take a look at an example that reads the content of the D:/learning/packt/resources.txt file and displays it on the screen (notice that we run the code in a *try-with-resources*, and so the file is closed by closing the stream):

private static final String FILE\_PATH   
 = "D:/learning/packt/resources.txt";  
...  
try (Stream<String> filesStream = Files.lines(  
 Paths.get(FILE\_PATH), StandardCharsets.UTF\_8)) {  
  
 filesStream.forEach(System.out::println);  
} catch (IOException e) {  
 // handle IOException if needed, otherwise remove the catch block  
}Copy

A similar method without arguments is available in the BufferedReader class:

try (BufferedReader brStream = Files.newBufferedReader(  
 Paths.get(FILE\_PATH), StandardCharsets.UTF\_8)) {  
  
 brStream.lines().forEach(System.out::println);  
} catch (IOException e) {  
 // handle IOException if needed, otherwise remove the catch block  
}Copy

137. Searching for files/folders in a file tree

Searching for files or folders in a file tree is a common task that's needed in a lot of situations. Thanks to JDK 8 and the new Files.find() method, we can accomplish this pretty easily.

The Files.find() method returns a Stream<Path> which is lazily populated with the paths that match the provided finding constraints:

public static Stream<Path> find​(  
 Path start,  
 int maxDepth,  
 BiPredicate<Path, ​BasicFileAttributes > matcher,  
 FileVisitOption...options  
) throws IOExceptionCopy

This method acts as the walk() method, and so it traverses the current file tree, starting from the given path (start), and reaching the maximum given depth (maxDepth). During the iteration of the current file tree, this method applies the given predicate (matcher). Via this predicate, we specify the constraints that must be matched by each file that goes in the final stream. Optionally, we can specify a set of visiting options (options).

Path startPath = Paths.get("D:/learning");Copy

Let's take a look at some examples that are meant to clarify the usage of this method:

* Find all the files ending with the .properties extension and follow the *symbolic links*:

Stream<Path> resultAsStream = Files.find(  
 startPath,  
 Integer.MAX\_VALUE,  
 (path, attr) -> path.toString().endsWith(".properties"),  
 FileVisitOption.FOLLOW\_LINKS  
);Copy

* Find all the regular files whose names start with application:

Stream<Path> resultAsStream = Files.find(  
 startPath,  
 Integer.MAX\_VALUE,  
 (path, attr) -> attr.isRegularFile() &amp;&amp;  
 path.getFileName().toString().startsWith("application")  
);Copy

* Find all the directories that were created after 16 March 2019:

Stream<Path> resultAsStream = Files.find(  
 startPath,  
 Integer.MAX\_VALUE,  
 (path, attr) -> attr.isDirectory() &amp;&amp;  
 attr.creationTime().toInstant()  
 .isAfter(LocalDate.of(2019, 3, 16).atStartOfDay()  
 .toInstant(ZoneOffset.UTC))  
);Copy

If we prefer to express the constraints as an expression (for example, a regular expression), then we can use the PathMatcher interface. This interface comes with a method called matches(Path path), which can tell if the given path matches this matcher's pattern.

A FileSystem implementation supports the *glob* and *regex* syntaxes (and may support others) via FileSystem.getPathMatcher(String syntaxPattern). The constraints take the form of syntax:pattern.

Based on PathMatcher, we can write helper methods that are capable of covering a wide range of constraints. For example, the following helper method only fetches files that respect the given constraint as a syntax:pattern:

public static Stream<Path> fetchFilesMatching(Path root,  
 String syntaxPattern) throws IOException {  
  
 final PathMatcher matcher  
 = root.getFileSystem().getPathMatcher(syntaxPattern);  
  
 return Files.find(root, Integer.MAX\_VALUE, (path, attr)  
 -> matcher.matches(path) &amp;&amp; !attr.isDirectory());  
}Copy

Finding all Java files via the *glob* syntax can be achieved as follows:

Stream<Path> resultAsStream   
 = fetchFilesMatching(startPath, "glob:\*\*/\*.java");Copy

If all we want to do is list the files from the current folder (without any constraints and a single level deep), then we can rely on the Files.list() method, as shown in the following example:

try (Stream<Path> allfiles = Files.list(startPath)) {  
 ...  
}Copy

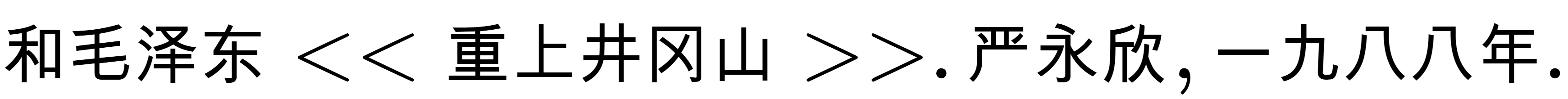
138. Reading/writing text files efficiently

In Java, reading files efficiently is a matter of choosing the right approach. For a better understanding of the following example, let's assume that our platform's default charset is UTF-8. Programmatically, the platform's default charset can be obtained via Charset.defaultCharset().

First, we need to distinguish between raw binary data and text files from a Java perspective. Dealing with raw binary data is the job of two abstract classes, that is, InputStream and OutputStream. For streaming files of raw binary data, we focus on the FileInputStream and FileOutputStream classes, which read/write a byte (8 bits) at a time. For famous types of binary data, we also have dedicated classes (for example, an audio file should be processed via AudioInputStream instead of FileInputStream).

While these classes are doing a spectacular job for raw binary data, they are not good for text files because they are slow and may produce wrong outputs. This becomes pretty clear if we think that streaming a text file via these classes means that each byte is read from the text file and processed (the same tedious flow is needed for writing a byte). Moreover, if a char has more than 1 byte, then it is possible to see some weird characters. In other words, decoding and encoding 8 bits independent of the charset (for example, Latin, Chinese, and so on) may produce unexpected output.

For example, let's suppose that we have the following Chinese poem saved in UTF-16:

Path chineseFile = Paths.get("chinese.txt");  
  
Forma

Descripción generada automáticamente con confianza media  
Forma

Descripción generada automáticamente con confianza media  
Forma

Descripción generada automáticamente con confianza media  
...Copy

The following code will not display it as expected:

try (InputStream is = new FileInputStream(chineseFile.toString())) {  
  
 int i;  
 while ((i = is.read()) != -1) {  
 System.out.print((char) i);  
 }  
}Copy

So, in order to fix this, we should specify the proper charset. While InputStream doesn't have support for this, we can rely on InputStreamReader (or OutputStreamReader, respectively). This class is a bridge from raw byte streams to character streams and allows us to specify the charset:

try (InputStreamReader isr = new InputStreamReader(  
 new FileInputStream(chineseFile.toFile()),   
 StandardCharsets.UTF\_16)) {  
  
 int i;  
 while ((i = isr.read()) != -1) {  
 System.out.print((char) i);  
 }  
}Copy

Things are back on track but are still slow! Now, the application can read more than one single byte at once (depending on the charset) and decodes them into characters using the specified charset. But a few more bytes are still slow.

InputStreamReader is a bridge between ray binary data streams and character streams. But Java provides the FileReader class as well. Its goal is to eliminate this bridge for character streams that are represented by character files.

For text files, we have a dedicated class known as the FileReader class (or FileWriter, respectively). This class reads 2 or 4 bytes (depending on the used charset) at a time. Actually, before JDK 11, FileReader didn't support an explicit charset. It simply used the platform's default charset. This isn't good for us because the following code will not produce the expected output:

try (FileReader fr = new FileReader(chineseFile.toFile())) {  
  
 int i;  
 while ((i = fr.read()) != -1) {  
 System.out.print((char) i);  
 }  
}Copy

But starting with JDK 11, the FileReader class was enriched with two more constructors that support an explicit charset:

* FileReader​(File file, Charset charset)
* FileReader​(String fileName, Charset charset)

This time, we can rewrite the preceding snippet of code and obtain the expected output:

try (FileReader frch = new FileReader(  
 chineseFile.toFile(), StandardCharsets.UTF\_16)) {  
  
 int i;  
 while ((i = frch.read()) != -1) {  
 System.out.print((char) i);  
 }  
}Copy

Reading 2 or 4 bytes at a time is still better than reading 1, but it's still slow. Moreover, notice that the preceding solutions use an int to store the retrieved char, and we need to explicitly cast it to char in order to display it. Basically, the retrieved char from the input file is converted into an int, and we convert it back into a char.

This is where *buffering streams* enter the scene. Think about what happens when we watch a video online. While we are watching the video, the browser is buffering the incoming bytes ahead of time. This way, we have a smooth experience because we can see the bytes from the buffer and avoid the potential interruptions caused by seeing the bytes during network transfer:

Diagrama

Descripción generada automáticamente

The same principle is used by classes such as BufferedInputStream, BufferedOutputStream for raw binary streams and BufferedReader, and BufferedWriter for character streams. The main idea is to buffer the data before processing. This time, FileReader returns the data to BufferedReader until it hits the end of the line (for example, \n or \n\r). BufferedReader uses RAM to store the buffered data:

try (BufferedReader br = new BufferedReader(  
 new FileReader(chineseFile.toFile(), StandardCharsets.UTF\_16))) {  
  
 String line;  
 // keep buffering and print  
 while ((line = br.readLine()) != null) {  
 System.out.println(line);  
 }  
}Copy

So, instead of reading 2 bytes at a time, we read a complete line, which is much faster. This is a really efficient way of reading text files.

For further optimization, we can set the size of the buffer via dedicated constructors.

Notice that the BufferedReader class knows how to create and deal with the buffer in the context of the incoming data but is independent of the source of data. In our example, the source of data is FileReader, which is a file, but the same BufferedReader can buffer data from different sources (for example, network, file, console, printer, sensor, and so on). In the end, we read what we buffered.

The preceding examples represent the main approaches for reading text files in Java. Starting with JDK 8, a new set of methods were added to make our life easier. In order to create a BufferedReader, we can rely on Files.newBufferedReader​(Path path, Charset cs) as well:

try (BufferedReader br = Files.newBufferedReader(  
 chineseFile, StandardCharsets.UTF\_16)) {  
  
 String line;  
 while ((line = br.readLine()) != null) {  
 System.out.println(line);  
 }  
}Copy

For BufferedWriter, we have Files.newBufferedWriter(). The advantage of these methods is that they support Path directly.

For fetching a text file's content as a Stream<T>, take a look at the problem in the *Streaming a file's content* section.

Another valid solution that may cause eye strain is as follows:

try (BufferedReader br = new BufferedReader(new InputStreamReader(  
 new FileInputStream(chineseFile.toFile()),   
 StandardCharsets.UTF\_16))) {  
  
 String line;  
 while ((line = br.readLine()) != null) {  
 System.out.println(line);  
 }  
}Copy

Now, it's time to talk about reading text files directly into memory.

Reading text files in memory

The Files class comes with two methods that can read an entire text file in memory. One of them is List<String> readAllLines​(Path path, Charset cs):

List<String> lines = Files.readAllLines(  
 chineseFile, StandardCharsets.UTF\_16);Copy

Moreover, we can read the entire content in a String via Files.readString​(Path path, Charset cs):

String content = Files.readString(chineseFile,   
 StandardCharsets.UTF\_16);Copy

While these methods are very convenient for relatively small files, they are not a good choice for large files. Trying to fetch large files in memory is prone to OutOfMemoryError and, obviously, will consume a lot of memory. Alternatively, in the case of huge files (for example, 200 GB), we can focus on memory-mapped files (MappedByteBuffer). MappedByteBuffer allows us to create and modify huge files and treat them as very big arrays. They look like they are in memory, even if they are not. Everything happens at the native level:

// or use, Files.newByteChannel()  
try (FileChannel fileChannel = (FileChannel.open(chineseFile,  
 EnumSet.of(StandardOpenOption.READ)))) {  
  
 MappedByteBuffer mbBuffer = fileChannel.map(  
 FileChannel.MapMode.READ\_ONLY, 0, fileChannel.size());  
  
 if (mbBuffer != null) {  
 String bufferContent   
 = StandardCharsets.UTF\_16.decode(mbBuffer).toString();  
  
 System.out.println(bufferContent);  
 mbBuffer.clear();  
 }  
}Copy

For huge files, it is advisable to traverse the buffer with a fixed size, as follows:

private static final int MAP\_SIZE = 5242880; // 5 MB in bytes  
  
try (FileChannel fileChannel = (FileChannel.open(chineseFile,  
 EnumSet.of(StandardOpenOption.READ)))) {  
  
 int position = 0;  
 long length = fileChannel.size();  
  
 while (position < length) {  
 long remaining = length - position;  
 int bytestomap = (int) Math.min(MAP\_SIZE, remaining);  
  
 MappedByteBuffer mbBuffer = fileChannel.map(  
 MapMode.READ\_ONLY, position, bytestomap);  
  
 ... // do something with the current buffer  
  
 position += bytestomap;  
 }  
}Copy

JDK 13 prepares the release of non-volatile MappedByteBuffers. Stay tuned!

Writing text files

For each class/method dedicated to reading a text file (for example, BufferedReader and readString()) Java provides its counterpart for writing a text file (for example, BufferedWriter and writeString()). Here is an example of writing a text file via BufferedWriter:

Path textFile = Paths.get("sample.txt");  
  
try (BufferedWriter bw = Files.newBufferedWriter(  
 textFile, StandardCharsets.UTF\_8, StandardOpenOption.CREATE,   
 StandardOpenOption.WRITE)) {  
 bw.write("Lorem ipsum dolor sit amet, ... ");  
 bw.newLine();  
 bw.write("sed do eiusmod tempor incididunt ...");  
}Copy

A very handy method for writing an Iterable into a text file is Files.write​(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options). For example, let's write the content of a list into a text file (each element from the list is written on a line in the file):

List<String> linesToWrite = Arrays.asList("abc", "def", "ghi");  
Path textFile = Paths.get("sample.txt");  
Files.write(textFile, linesToWrite, StandardCharsets.UTF\_8,  
 StandardOpenOption.CREATE, StandardOpenOption.WRITE);Copy

Finally, to write a String to a file, we can rely on the Files.writeString​(Path path, CharSequence csq, OpenOption... options) method:

Path textFile = Paths.get("sample.txt");  
  
String lineToWrite = "Lorem ipsum dolor sit amet, ...";  
Files.writeString(textFile, lineToWrite, StandardCharsets.UTF\_8,  
 StandardOpenOption.CREATE, StandardOpenOption.WRITE);Copy

Via StandardOpenOption, we can control how the file is opened. In the preceding examples, the files were created if they didn't exist (CREATE) and they were opened for write access (WRITE). Many other options are available (for example, APPEND, DELETE\_ON\_CLOSE, and so on).

Finally, writing a text file via MappedByteBuffer can be accomplished as follows (this can be useful for writing huge text files):

Path textFile = Paths.get("sample.txt");  
CharBuffer cb = CharBuffer.wrap("Lorem ipsum dolor sit amet, ...");  
  
try (FileChannel fileChannel = (FileChannel) Files.newByteChannel(  
 textFile, EnumSet.of(StandardOpenOption.CREATE,  
 StandardOpenOption.READ, StandardOpenOption.WRITE))) {  
  
 MappedByteBuffer mbBuffer = fileChannel  
 .map(FileChannel.MapMode.READ\_WRITE, 0, cb.length());  
  
 if (mbBuffer != null) {  
 mbBuffer.put(StandardCharsets.UTF\_8.encode(cb));  
 }  
}Copy

139. Reading/writing binary files efficiently

In the previous problem, *Reading/writing text files efficiently*, we talked about *buffering streaming* (for a clear picture, consider reading that problem before this one). Things work the same for binary files too, and so we can jump directly into some examples.

Let's consider the following binary file and its size in bytes:

Path binaryFile = Paths.get(  
 "build/classes/modern/challenge/Main.class");  
  
int fileSize = (int) Files.readAttributes(  
 binaryFile, BasicFileAttributes.class).size();Copy

We can read the file's content in a byte[] via FileInputStream (this doesn't use buffering):

final byte[] buffer = new byte[fileSize];  
try (InputStream is = new FileInputStream(binaryFile.toString())) {  
  
 int i;  
 while ((i = is.read(buffer)) != -1) {  
 System.out.print("\nReading ... ");  
 }  
}Copy

However, the preceding example isn't very efficient. Achieving high efficiency when it comes to reading the buffer.length bytes from this input stream into a byte array can be done via BufferedInputStream, as follows:

final byte[] buffer = new byte[fileSize];  
  
try (BufferedInputStream bis = new BufferedInputStream(  
 new FileInputStream(binaryFile.toFile()))) {  
  
 int i;  
 while ((i = bis.read(buffer)) != -1) {  
 System.out.print("\nReading ... " + i);  
 }  
}Copy

FileInputStream can be obtained via the Files.newInputStream() method as well. The advantage of this method consists of the fact that it supports Path directly:

final byte[] buffer = new byte[fileSize];  
  
try (BufferedInputStream bis = new BufferedInputStream(  
 Files.newInputStream(binaryFile))) {  
  
 int i;  
 while ((i = bis.read(buffer)) != -1) {  
 System.out.print("\nReading ... " + i);  
 }  
}Copy

If the file is too large to fit in a buffer of the file size, then it is preferable to read it via a smaller buffer with a fixed size (for example, 512 bytes) and the read() flavors, which are as follows:

* read​(byte[] b)
* read​(byte[] b, int off, int len)
* readNBytes​(byte[] b, int off, int len)
* readNBytes​(int len)

The read() method without arguments will read the input stream byte by byte. This is the most inefficient way, especially without using buffering.

Alternatively, if our goal is to read the input stream as a byte array, we can rely on ByteArrayInputStream (it uses an internal buffer, so there is no need to use BufferedInputStream):

final byte[] buffer = new byte[fileSize];  
  
try (ByteArrayInputStream bais = new ByteArrayInputStream(buffer)) {  
  
 int i;  
 while ((i = bais.read(buffer)) != -1) {  
 System.out.print("\nReading ... ");  
 }  
}Copy

The preceding approaches are a good fit for raw binary data, but sometimes, our binary files contain certain data (for example, ints, floats, and so on). In such cases, DataInputStream and DataOutputStream provide convenient methods for reading and writing certain data types. Let's consider that we have a file, data.bin, that contains float numbers. We can efficiently read it as follows:

Path dataFile = Paths.get("data.bin");  
  
try (DataInputStream dis = new DataInputStream(  
 new BufferedInputStream(Files.newInputStream(dataFile)))) {  
  
 while (dis.available() > 0) {  
 float nr = dis.readFloat();  
 System.out.println("Read: " + nr);  
 }  
}Copy

These two classes are just two of the *data filters* provided by Java. For an overview of all the supported *data filters*, check out the subclasses of FilterInputStream. Moreover, the Scanner class is a good alternative for reading certain types of data. Check out the problem in the *Working with Scanner* section for more information.

Now, let's see how we can read binary files directly into memory.

Reading binary files into memory

Reading an entire binary file into memory can be accomplished via Files.readAllBytes():

byte[] bytes = Files.readAllBytes(binaryFile);Copy

A similar method exists in the InputStream class as well.

While these methods are very convenient for relatively small files, they are not a good choice for large files. Trying to fetch large files into memory is prone to OOM errors and, obviously, will consume a lot of memory. Alternatively, in the case of huge files (e.g., 200 GB), we can focus on memory-mapped files (MappedByteBuffer). MappedByteBuffer allows us to create and modify huge files and treat them as a very big array. They look like they are in memory even if they are not. Everything happens at the native level:

try (FileChannel fileChannel = (FileChannel.open(binaryFile,  
 EnumSet.of(StandardOpenOption.READ)))) {  
  
 MappedByteBuffer mbBuffer = fileChannel.map(  
 FileChannel.MapMode.READ\_ONLY, 0, fileChannel.size());  
  
 System.out.println("\nRead: " + mbBuffer.limit() + " bytes");  
}Copy

For huge files it is advisable to traverse the buffer with a fixed size as follows:

private static final int MAP\_SIZE = 5242880; // 5 MB in bytes  
  
try (FileChannel fileChannel = FileChannel.open(  
 binaryFile, StandardOpenOption.READ)) {  
  
 int position = 0;  
 long length = fileChannel.size();  
  
 while (position < length) {  
 long remaining = length - position;  
 int bytestomap = (int) Math.min(MAP\_SIZE, remaining);  
  
 MappedByteBuffer mbBuffer = fileChannel.map(  
 MapMode.READ\_ONLY, position, bytestomap);  
  
 ... // do something with the current buffer  
  
 position += bytestomap;  
 }  
}Copy

Writing binary files

An efficient way of writing binary files is by using BufferedOutputStream. For example, writing a byte[] to a file can be accomplished as follows:

final byte[] buffer...;  
Path classFile = Paths.get(  
 "build/classes/modern/challenge/Main.class");  
  
try (BufferedOutputStream bos = newBufferedOutputStream(  
 Files.newOutputStream(classFile, StandardOpenOption.CREATE,  
 StandardOpenOption.WRITE))) {  
  
 bos.write(buffer);  
}Copy

If you're writing byte by byte, use the write(int b) method, and, if you're writing a chunk of data, use the write​(byte[] b, int off, int len) method.

A very handy method for writing a byte[] to a file is Files.write​(Path path, byte[] bytes, OpenOption... options). For example, let's write the content of the preceding buffer:

Path classFile = Paths.get(  
 "build/classes/modern/challenge/Main.class");  
  
Files.write(classFile, buffer,  
 StandardOpenOption.CREATE, StandardOpenOption.WRITE);Copy

Writing a binary file via MappedByteBuffer can be accomplished as follows (this can be useful for writing huge text files):

Path classFile = Paths.get(  
 "build/classes/modern/challenge/Main.class");  
try (FileChannel fileChannel = (FileChannel) Files.newByteChannel(  
 classFile, EnumSet.of(StandardOpenOption.CREATE,  
 StandardOpenOption.READ, StandardOpenOption.WRITE))) {  
  
 MappedByteBuffer mbBuffer = fileChannel  
 .map(FileChannel.MapMode.READ\_WRITE, 0, buffer.length);  
  
 if (mbBuffer != null) {  
 mbBuffer.put(buffer);  
 }  
}Copy

Finally, if we are writing a certain piece of data (not raw binary data), then we can rely on DataOutputStream. This class comes with write*Foo*() methods for different kinds of data. For example, let's write several floats into a file:

Path floatFile = Paths.get("float.bin");  
  
try (DataOutputStream dis = new DataOutputStream(  
 new BufferedOutputStream(Files.newOutputStream(floatFile)))) {  
 dis.writeFloat(23.56f);  
 dis.writeFloat(2.516f);  
 dis.writeFloat(56.123f);  
}Copy

140. Searching in big files

Searching and counting the number of occurrences of a certain string in a file is a common task. Trying to achieve this as fast as possible is a mandatory requirement, especially if the file is big (for example, 200 GB).

Note that the following implementations assume that string *11* occurs only once in *111*, not twice. Moreover, the first three implementations rely on the following helper method from [Chapter 1](https://subscription.packtpub.com/book/programming/9781789801415/6/ch06lvl1sec24/e9e220b6-0de5-4d1c-93f6-19b027350f7c.xhtml), *Strings, Numbers, and Math*, the *Counting a string in another string* section:

private static int countStringInString(String string, String tofind) {  
 return string.split(Pattern.quote(tofind), -1).length - 1;  
}Copy

With that being said, let's take a look at several approaches to this problem.

Solution based on BufferedReader

We already know from the previous problems that BufferedReader is very efficient for reading text files. Therefore, we can use it to read a big file as well. While reading, for each line obtained via BufferedReader.readLine(), we need to count the number of occurrences of the searched string via countStringInString():

public static int countOccurrences(Path path, String text, Charset ch)  
 throws IOException {  
  
 int count = 0;  
  
 try (BufferedReader br = Files.newBufferedReader(path, ch)) {  
 String line;  
 while ((line = br.readLine()) != null) {  
 count += countStringInString(line, text);  
 }  
 }  
  
 return count;  
}Copy

Solution based on Files.readAllLines()

If memory (RAM) is not a problem for us, then we can try to read the entire file into memory (via Files.readAllLines()) and process it from there. Having the entire file in memory sustains parallel processing. Therefore, if our hardware can be highlighted by parallel processing, then we can try to rely on parallelStream(), as follows:

public static int countOccurrences(Path path, String text, Charset ch)  
 throws IOException {  
  
 return Files.readAllLines(path, ch).parallelStream()  
 .mapToInt((p) -> countStringInString(p, text))  
 .sum();  
}Copy

If parallelStream() doesn't come with any benefits, then we can simply switch to stream(). It is just a matter of benchmarking.

Solution based on Files.lines()

We can try to take advantage of streams via Files.lines() as well. This time, we fetch the file as a lazy Stream<String>. If we can take advantage of parallel processing (benchmarking reveals better performances), then it is very simple to parallelize Stream<String> by calling the parallel() method:

public static int countOccurrences(Path path, String text, Charset ch)  
 throws IOException {  
  
 return Files.lines(path, ch).parallel()  
 .mapToInt((p) -> countStringInString(p, text))  
 .sum();  
}Copy

Solution based on Scanner

Starting with JDK 9, the Scanner class comes with a method that returns a stream of delimiter-separated tokens, Stream<String> tokens(). If we treat the text to search as the delimiter of Scanner and we count the entries of the Stream returned by tokens(), then we obtain the correct result:

public static long countOccurrences(  
 Path path, String text, Charset ch) throws IOException {  
  
 long count;  
  
 try (Scanner scanner = new Scanner(path, ch)  
 .useDelimiter(Pattern.quote(text))) {  
  
 count = scanner.tokens().count() - 1;  
 }  
  
 return count;  
}Copy

The constructors for scanner that support an explicit charset were added in JDK 10.

Solution based on MappedByteBuffer

The last solution that we'll talk about here is based on Java NIO.2, MappedByteBuffer, and FileChannel. This solution opens a memory-mapped byte buffer (MappedByteBuffer) from a FileChannel on the given file. We traverse the fetched byte buffer and look for matches with the searched string (this string is converted into a byte[] and searching take place byte by byte).

For small files, it is faster to load the entire file into memory. For large/huge files, it is faster to load and process the files in chunks (for example, a chunk of 5 MB). Once we have loaded a chunk, we have to count the number of occurrences of the searched string. We store the result and pass it to the next chunk of data. We repeat this until the entire file has been traversed.

Let's take a look at the core lines of this implementation (take a look at the source code bundled with this book for the complete code):

private static final int MAP\_SIZE = 5242880; // 5 MB in bytes  
  
public static int countOccurrences(Path path, String text)  
 throws IOException {  
  
 final byte[] texttofind = text.getBytes(StandardCharsets.UTF\_8);  
 int count = 0;  
  
 try (FileChannel fileChannel = FileChannel.open(path,  
 StandardOpenOption.READ)) {  
 int position = 0;  
 long length = fileChannel.size();  
  
 while (position < length) {  
 long remaining = length - position;  
 int bytestomap = (int) Math.min(MAP\_SIZE, remaining);  
  
 MappedByteBuffer mbBuffer = fileChannel.map(  
 MapMode.READ\_ONLY, position, bytestomap);  
  
 int limit = mbBuffer.limit();  
 int lastSpace = -1;  
 int firstChar = -1;  
  
 while (mbBuffer.hasRemaining()) {   
 // spaghetti code omitted for brevity  
 ...  
 }  
 }  
 }  
  
 return count;  
}Copy

This solution is extremely fast because the file is read directly from the operating system's memory without having to be loaded into the JVM. The operations take place at the native level, called the operating system level. Note that this implementation works only for the UTF-8 charset, but it can be adapted for other charsets as well.

141. Reading a JSON/CSV file as an object

JSON and CSV files are everywhere these days. Reading (deserialize) JSON/CSV files can be a day-to-day task that typically precedes our business logic. Writing (serialize) JSON/CSV files is also a popular task that typically occurs at the end of the business logic. Between reading and writing such files, an application uses the data as objects.

Read/write a JSON file as an object

Let's start with three text files that represent typical JSON-like mappings:

Texto

Descripción generada automáticamente

In melons\_raw.json, we have a JSON entry per line. Each line is a piece of JSON that's independent of the previous line but has the same schema. In melons\_array.json, we have a JSON array, and in melons\_map.json, we have a JSON that fits well in a Java Map.

For each of these files, we have a Path, as follows:

Path pathArray = Paths.get("melons\_array.json");  
Path pathMap = Paths.get("melons\_map.json");  
Path pathRaw = Paths.get("melons\_raw.json");Copy

Now, let's take a look at three dedicated libraries for reading the contents of these files as Melon instances:

public class Melon {  
  
 private String type;  
 private int weight;  
  
 // getters and setters omitted for brevity  
}Copy

Using JSON-B

Java EE 8 comes with a JAXB-like, declarative JSON binding called JSON-B (JSR-367). JSON-B is consistent with JAXB and other Java EE/SE APIs. Jakarta EE takes Java EE 8 JSON (P and B) to the next level. Its API is exposed via the javax.json.bind.Jsonb and javax.json.bind.JsonbBuilder classes:

Jsonb jsonb = JsonbBuilder.create();Copy

For deserialization, we use Jsonb.fromJson(), while, for serialization, we use Jsonb.toJson():

* Let's read melons\_array.json as an Array of Melon:

Melon[] melonsArray = jsonb.fromJson(Files.newBufferedReader(  
 pathArray, StandardCharsets.UTF\_8), Melon[].class);Copy

* Let's read melons\_array.json as a List of Melon:

List<Melon> melonsList   
 = jsonb.fromJson(Files.newBufferedReader(  
 pathArray, StandardCharsets.UTF\_8), ArrayList.class);Copy

* Let's read melons\_map.json as a Map of Melon:

Map<String, Melon> melonsMap   
 = jsonb.fromJson(Files.newBufferedReader(  
 pathMap, StandardCharsets.UTF\_8), HashMap.class);Copy

* Let's read melons\_raw.json line by line into a Map:

Map<String, String> stringMap = new HashMap<>();  
  
try (BufferedReader br = Files.newBufferedReader(  
 pathRaw, StandardCharsets.UTF\_8)) {  
  
 String line;  
  
 while ((line = br.readLine()) != null) {  
 stringMap = jsonb.fromJson(line, HashMap.class);  
 System.out.println("Current map is: " + stringMap);  
 }  
}Copy

* Let's read melons\_raw.json line by line into a Melon:

try (BufferedReader br = Files.newBufferedReader(  
 pathRaw, StandardCharsets.UTF\_8)) {  
  
 String line;  
  
 while ((line = br.readLine()) != null) {  
 Melon melon = jsonb.fromJson(line, Melon.class);  
 System.out.println("Current melon is: " + melon);  
 }  
}Copy

* Let's write an object into a JSON file (melons\_output.json):

Path path = Paths.get("melons\_output.json");  
  
jsonb.toJson(melonsMap, Files.newBufferedWriter(path,  
 StandardCharsets.UTF\_8, StandardOpenOption.CREATE,   
 StandardOpenOption.WRITE));Copy

Using Jackson

Jackson is a popular and fast library dedicated to processing (serializing/deserializing) JSON data. The Jackson API relies on com.fasterxml.jackson.databind.ObjectMapper. Let's go over the preceding examples again, but this time using Jackson:

ObjectMapper mapper = new ObjectMapper();Copy

For deserialization, we use ObjectMapper.readValue(), while for serialization, we use ObjectMapper.writeValue():

* Let's read melons\_array.json as an Array of Melon:

Melon[] melonsArray   
 = mapper.readValue(Files.newBufferedReader(  
 pathArray, StandardCharsets.UTF\_8), Melon[].class);Copy

* Let's read melons\_array.json as a List of Melon:

List<Melon> melonsList   
 = mapper.readValue(Files.newBufferedReader(  
 pathArray, StandardCharsets.UTF\_8), ArrayList.class);Copy

* Let's read melons\_map.json as a Map of Melon:

Map<String, Melon> melonsMap   
 = mapper.readValue(Files.newBufferedReader(  
 pathMap, StandardCharsets.UTF\_8), HashMap.class);Copy

* Let's read melons\_raw.json line by line into a Map:

Map<String, String> stringMap = new HashMap<>();  
  
try (BufferedReader br = Files.newBufferedReader(  
 pathRaw, StandardCharsets.UTF\_8)) {  
  
 String line;  
  
 while ((line = br.readLine()) != null) {  
 stringMap = mapper.readValue(line, HashMap.class);  
 System.out.println("Current map is: " + stringMap);  
 }  
}Copy

* Let's read melons\_raw.json line by line into a Melon:

try (BufferedReader br = Files.newBufferedReader(  
 pathRaw, StandardCharsets.UTF\_8)) {  
  
 String line;  
  
 while ((line = br.readLine()) != null) {  
 Melon melon = mapper.readValue(line, Melon.class);  
 System.out.println("Current melon is: " + melon);  
 }  
}Copy

* Let's write an object into a JSON file (melons\_output.json):

Path path = Paths.get("melons\_output.json");  
  
mapper.writeValue(Files.newBufferedWriter(path,   
 StandardCharsets.UTF\_8, StandardOpenOption.CREATE,   
 StandardOpenOption.WRITE), melonsMap);Copy

Using Gson

Gson is another fast library dedicated to processing (serializing/deserializing) JSON data. In a Maven project, it can be added as a dependency in pom.xml. Its API relies on a class name, com.google.gson.Gson. The code that's bundled with this book provides a suite of examples for it.

Reading a CSV file as an object

The simplest CSV file looks like the file in the following illustration (lines of data separated by commas):

Texto

Descripción generada automáticamente

A simple and efficient solution to deserializing this kind of CSV file relies on the BufferedReader and String.split() methods. We can read each line from the file via BufferedReader.readLine() and split it with a comma delimiter via Spring.split(). The result (each line of content) can be stored in a List<String>. The final result will be a List<List<String>>, as follows:

public static List<List<String>> readAsObject(  
 Path path, Charset cs, String delimiter) throws IOException {  
  
 List<List<String>> content = new ArrayList<>();  
  
 try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
  
 String line;  
  
 while ((line = br.readLine()) != null) {  
 String[] values = line.split(delimiter);  
 content.add(Arrays.asList(values));  
 }  
 }  
  
 return content;  
}Copy

If the CSV data has POJOs correspondents (for example, our CSV is the result of serializing a bunch of Melon instances), then it can be deserialized, as shown in the following example:

public static List<Melon> readAsMelon(  
 Path path, Charset cs, String delimiter) throws IOException {  
  
 List<Melon> content = new ArrayList<>();  
  
 try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
  
 String line;  
  
 while ((line = br.readLine()) != null) {  
 String[] values = line.split(Pattern.quote(delimiter));  
 content.add(new Melon(values[0], Integer.valueOf(values[1])));  
 }  
 }  
  
 return content;  
}Copy

For complex CSV files, it is advisable to rely on dedicated libraries (for example, OpenCSV, Apache Commons CSV, Super CSV, and so on).

142. Working with temporary files/folders

The Java NIO.2 API provides support for working with temporary folders/files. For example, we can easily locate the default location for temporary folders/files as follows:

String defaultBaseDir = System.getProperty("java.io.tmpdir");Copy

Commonly, in Windows, the default temporary folder is C:\Temp, %Windows%\Temp, or a temporary directory per user in Local Settings\Temp (this location is usually controlled via the TEMP environment variable). In Linux/Unix, the global temporary directories are /tmp and /var/tmp. The preceding line of code will return the default location, depending on the operating system.

In the next section, we'll learn how to create a temporary folder/file.

Creating a temporary folder/file

Creating a temporary folder can be accomplished using Path createTempDirectory​(Path dir, String prefix, FileAttribute<?>... attrs). This is a static method in the Files class that can be used as follows:

* Let's create a temporary folder in the OS's default location with no prefix:

// C:\Users\Anghel\AppData\Local\Temp\8083202661590940905  
Path tmpNoPrefix = Files.createTempDirectory(null);Copy

* Let's create a temporary folder in the OS's default location with a custom prefix:

// C:\Users\Anghel\AppData\Local\Temp\logs\_5825861687219258744  
String customDirPrefix = "logs\_";  
Path tmpCustomPrefix   
 = Files.createTempDirectory(customDirPrefix);Copy

* Let's create a temporary folder in a custom location with a custom prefix:

// D:\tmp\logs\_10153083118282372419  
Path customBaseDir   
 = FileSystems.getDefault().getPath("D:/tmp");  
String customDirPrefix = "logs\_";  
Path tmpCustomLocationAndPrefix   
 = Files.createTempDirectory(customBaseDir, customDirPrefix);Copy

Creating a temporary file can be accomplished via Path createTempFile​(Path dir, String prefix, String suffix, FileAttribute<?>... attrs). This is a static method in the Files class that can be used as follows:

* Let's create a temporary file in the OS's default location with no prefix and suffix:

// C:\Users\Anghel\AppData\Local\Temp\16106384687161465188.tmp  
Path tmpNoPrefixSuffix = Files.createTempFile(null, null);Copy

* Let's create a temporary file in the OS's default location with a custom prefix and suffix:

// C:\Users\Anghel\AppData\Local\Temp\log\_402507375350226.txt  
String customFilePrefix = "log\_";  
String customFileSuffix = ".txt";  
Path tmpCustomPrefixAndSuffix   
 = Files.createTempFile(customFilePrefix, customFileSuffix);Copy

* Let's create a temporary file in a custom location with a custom prefix and suffix:

// D:\tmp\log\_13299365648984256372.txt  
Path customBaseDir   
 = FileSystems.getDefault().getPath("D:/tmp");  
String customFilePrefix = "log\_";  
String customFileSuffix = ".txt";  
Path tmpCustomLocationPrefixSuffix = Files.createTempFile(  
 customBaseDir, customFilePrefix, customFileSuffix);Copy

In the following sections, we'll take a look at the different ways we can delete a temporary folder/file.

Deleting a temporary folder/file via shutdown-hook

Deleting a temporary folder/file is a task that can be accomplished by the operating system or specialized tools. However, sometimes, we need to control this programmatically and delete a folder/file based on different design considerations.

A solution to this problem relies on the *shutdown-hook* mechanism, which can be implemented via the Runtime.getRuntime().addShutdownHook() method. This mechanism is useful whenever we need to complete certain tasks (for example, cleanup tasks) right before the JVM shuts down. It is implemented as a Java thread whose run() method is invoked when the *shutdown-hook* is executed by JVM at shut down. This is shown in the following code:

Path customBaseDir = FileSystems.getDefault().getPath("D:/tmp");  
String customDirPrefix = "logs\_";  
String customFilePrefix = "log\_";  
String customFileSuffix = ".txt";  
  
try {  
 Path tmpDir = Files.createTempDirectory(  
 customBaseDir, customDirPrefix);  
 Path tmpFile1 = Files.createTempFile(  
 tmpDir, customFilePrefix, customFileSuffix);  
 Path tmpFile2 = Files.createTempFile(  
 tmpDir, customFilePrefix, customFileSuffix);  
  
 Runtime.getRuntime().addShutdownHook(new Thread() {  
 @Override  
 public void run() {  
 try (DirectoryStream<Path> ds   
 = Files.newDirectoryStream(tmpDir)) {  
 for (Path file: ds) {  
 Files.delete(file);  
 }  
  
 Files.delete(tmpDir);  
 } catch (IOException e) {  
 ...  
 }  
 }  
 });  
  
 //simulate some operations with temp file until delete it  
 Thread.sleep(10000);  
} catch (IOException | InterruptedException e) {  
 ...  
}Copy

A *shutdown-hook* will not be executed in the case of abnormal/forced terminations (for example, JVM crashes, Terminal operations are triggered, and so on). It runs when all the threads finish or when System.exit(0) is called. It is advisable to run it fast since they can be forcibly stopped before completion if something goes wrong (for example, the OS shuts down). Programmatically, a *shutdown-hook* can only be stopped by Runtime.halt().

Deleting a temporary folder/file via deleteOnExit()

Another solution for deleting a temporary folder/file relies on the File.deleteOnExit() method. By calling this method, we can register for the deletion of a folder/file. The deletion action happens when JVM shuts down:

Path customBaseDir = FileSystems.getDefault().getPath("D:/tmp");  
String customDirPrefix = "logs\_";  
String customFilePrefix = "log\_";  
String customFileSuffix = ".txt";  
  
try {  
 Path tmpDir = Files.createTempDirectory(  
 customBaseDir, customDirPrefix);  
 System.out.println("Created temp folder as: " + tmpDir);  
 Path tmpFile1 = Files.createTempFile(  
 tmpDir, customFilePrefix, customFileSuffix);  
 Path tmpFile2 = Files.createTempFile(  
 tmpDir, customFilePrefix, customFileSuffix);  
  
 try (DirectoryStream<Path> ds = Files.newDirectoryStream(tmpDir)) {  
 tmpDir.toFile().deleteOnExit();  
  
 for (Path file: ds) {  
 file.toFile().deleteOnExit();  
 }  
 } catch (IOException e) {  
 ...  
 }  
  
 // simulate some operations with temp file until delete it  
 Thread.sleep(10000);  
} catch (IOException | InterruptedException e) {  
 ...  
}Copy

It is advisable to only rely on this method (deleteOnExit()) when the application manages a small number of temporary folders/files. This method may consume a lot of memory (it consumes memory for each temporary resource that's registered for deletion) and this memory may not be released until JVM terminates. Pay attention, since this method needs to be called in order to register each temporary resource, and the deletion takes place in reverse order of registration (for example, we must register a temporary folder before registering its content).

Deleting a temporary file via DELETE\_ON\_CLOSE

Another solution when it comes to deleting a temporary file relies on StandardOpenOption.DELETE\_ON\_CLOSE (this deletes the file when the stream is closed). For example, the following piece of code creates a temporary file via the createTempFile() method and opens a buffered writer stream for it with DELETE\_ON\_CLOSE explicitly specified:

Path customBaseDir = FileSystems.getDefault().getPath("D:/tmp");  
String customFilePrefix = "log\_";  
String customFileSuffix = ".txt";  
Path tmpFile = null;  
  
try {  
 tmpFile = Files.createTempFile(  
 customBaseDir, customFilePrefix, customFileSuffix);  
} catch (IOException e) {  
 ...  
}  
  
try (BufferedWriter bw = Files.newBufferedWriter(tmpFile,  
 StandardCharsets.UTF\_8, StandardOpenOption.DELETE\_ON\_CLOSE)) {  
  
 //simulate some operations with temp file until delete it  
 Thread.sleep(10000);  
} catch (IOException | InterruptedException e) {  
 ...  
}Copy

This solution can be adopted for any file. It is not specific to temporary resources.

143. Filtering files

Filtering files from a Path is a very common task. For example, we may only want the files of a specific type, with a certain name pattern, modified today, and so on.

Filtering via Files.newDirectoryStream()

Without any kind of filter, we can easily loop a folder's content (one level deep) via the Files.newDirectoryStream(Path dir) method. This method returns a DirectoryStream<Path>, which is an object that we can use to iterate over the entries in a directory:

Path path = Paths.get("D:/learning/books/spring");  
  
try (DirectoryStream<Path> ds = Files.newDirectoryStream(path)) {  
  
 for (Path file: ds) {  
 System.out.println(file.getFileName());  
 }  
}Copy

If we want to enrich this snippet of code with a filter, then we have at least two solutions. One solution relies on another flavor of the newDirectoryStream() method, newDirectoryStream​(Path dir, String glob). Besides Path, this method receives a filter by using the *glob* syntax. For example, we can filter the D:/learning/books/spring folder for files that are of the PNG, JPG, and BMP types:

try (DirectoryStream<Path> ds =  
 Files.newDirectoryStream(path, "\*.{png,jpg,bmp}")) {  
  
 for (Path file: ds) {  
 System.out.println(file.getFileName());  
 }  
}Copy

When *glob* syntax cannot help us anymore, it's time to use another flavor of newDirectoryStream() that gets a Filter, that is, newDirectoryStream​(Path dir, DirectoryStream.Filter<? super Path> filter). First, let's define a filter for files larger than 10 MB:

DirectoryStream.Filter<Path> sizeFilter   
 = new DirectoryStream.Filter<>() {  
  
 @Override  
 public boolean accept(Path path) throws IOException {  
 return (Files.size(path) > 1024 \* 1024 \* 10);  
 }  
};Copy

We can also do this in functional-style:

DirectoryStream.Filter<Path> sizeFilter   
 = p -> (Files.size(p) > 1024 \* 1024 \* 10);Copy

Now, we can apply this filter like so:

try (DirectoryStream<Path> ds =  
 Files.newDirectoryStream(path, sizeFilter)) {  
  
 for (Path file: ds) {  
 System.out.println(file.getFileName() + " " +  
 Files.readAttributes(file, BasicFileAttributes.class).size()   
 + " bytes");  
 }  
}Copy

Let's check out a few more filters that we can use with this technique:

* The following is a user-defined filter for folders:

DirectoryStream.Filter<Path> folderFilter   
 = new DirectoryStream.Filter<>() {  
  
 @Override  
 public boolean accept(Path path) throws IOException {  
 return (Files.isDirectory(path, NOFOLLOW\_LINKS));  
 }  
};Copy

* The following is a user-defined filter for files that have been modified today:

DirectoryStream.Filter<Path> todayFilter   
 = new DirectoryStream.Filter<>() {  
  
 @Override  
 public boolean accept(Path path) throws IOException {  
 FileTime lastModified = Files.readAttributes(path,  
 BasicFileAttributes.class).lastModifiedTime();  
  
 LocalDate lastModifiedDate = lastModified.toInstant()  
 .atOffset(ZoneOffset.UTC).toLocalDate();  
 LocalDate todayDate = Instant.now()  
 .atOffset(ZoneOffset.UTC).toLocalDate();  
  
 return lastModifiedDate.equals(todayDate);  
 }  
};Copy

* The following is a user-defined filter for hidden files/folders:

DirectoryStream.Filter<Path> hiddenFilter   
 = new DirectoryStream.Filter<>() {  
  
 @Override  
 public boolean accept(Path path) throws IOException {  
 return (Files.isHidden(path));  
 }  
};Copy

In the following sections, we'll take a look at the different ways we can filter a file.

Filtering via FilenameFilter

The FilenameFilter functional interface can be used to filter files from a folder as well. First, we need to define a filter (for example, the following is a filter for files of the PDF type):

String[] files = path.toFile().list(new FilenameFilter() {  
  
 @Override  
 public boolean accept(File folder, String fileName) {  
 return fileName.endsWith(".pdf");  
 }  
});Copy

We can do the same in functional-style:

FilenameFilter filter = (File folder, String fileName)   
 -> fileName.endsWith(".pdf");Copy

Let's make this more concise:

FilenameFilter filter = (f, n) -> n.endsWith(".pdf");Copy

In order to use this filter, we need to pass it to the overloaded File.list​(FilenameFilter filter) or File.listFiles​(FilenameFilter filter) method:

String[] files = path.toFile().list(filter);Copy

The files array will only contain the names of the PDF files.

For fetching the result as a File[], we should call listFiles() instead of list().

Filtering via FileFilter

FileFilter is another functional interface that can be used to filter files and folders. For example, let's filter only folders:

File[] folders = path.toFile().listFiles(new FileFilter() {  
  
 @Override  
 public boolean accept(File file) {  
 return file.isDirectory();  
 }  
});Copy

We can do the same in functional-style:

File[] folders = path.toFile().listFiles((File file)   
 -> file.isDirectory());Copy

Let's make this more concise:

File[] folders = path.toFile().listFiles(f -> f.isDirectory());Copy

Finally, we can do this via member reference:

File[] folders = path.toFile().listFiles(File::isDirectory);Copy

144. Discovering mismatches between two files

The solution to this problem is comparing the content of two files (a byte by byte comparison) until the first mismatch is found or the EOF is reached.

Let's consider the following four text files:

Texto, Tabla

Descripción generada automáticamente

Only the first two files (file1.txt and file2.txt) are identical. Any other comparison should reveal the presence of at least one mismatch.

One solution is to use MappedByteBuffer. This solution is super-fast and easy to implement. We just open two FileChannels (one for each file) and perform a byte by byte comparison until we find the first mismatch or EOF. If the files don't have the same length in terms of bytes, then we assume that the files are not the same and return immediately:

private static final int MAP\_SIZE = 5242880; // 5 MB in bytes  
  
public static boolean haveMismatches(Path p1, Path p2)   
 throws IOException {  
  
 try (FileChannel channel1 = (FileChannel.open(p1,  
 EnumSet.of(StandardOpenOption.READ)))) {  
  
 try (FileChannel channel2 = (FileChannel.open(p2,  
 EnumSet.of(StandardOpenOption.READ)))) {  
  
 long length1 = channel1.size();  
 long length2 = channel2.size();  
  
 if (length1 != length2) {  
 return true;  
 }  
  
 int position = 0;  
 while (position < length1) {  
 long remaining = length1 - position;  
 int bytestomap = (int) Math.min(MAP\_SIZE, remaining);  
  
 MappedByteBuffer mbBuffer1 = channel1.map(  
 MapMode.READ\_ONLY, position, bytestomap);  
 MappedByteBuffer mbBuffer2 = channel2.map(  
 MapMode.READ\_ONLY, position, bytestomap);  
  
 while (mbBuffer1.hasRemaining()) {  
 if (mbBuffer1.get() != mbBuffer2.get()) {  
 return true;  
 }  
 }  
  
 position += bytestomap;  
 }  
 }  
 }  
  
 return false;  
}Copy

JDK 13 has prepared the release of non-volatile MappedByteBuffers. Stay tuned!

Starting with JDK 12, the Files class has been enriched with a new method dedicated to pointing mismatches between two files. This method has the following signature:

public static long mismatch​(Path path, Path path2) throws IOExceptionCopy

This method finds and returns the position of the first mismatched byte in the content of two files. If there is no mismatch, then it returns -1:

long mismatches12 = Files.mismatch(file1, file2); // -1  
long mismatches13 = Files.mismatch(file1, file3); // 51  
long mismatches14 = Files.mismatch(file1, file4); // 60Copy

145. Circular byte buffer

The Java NIO.2 API comes with an implementation of a byte buffer called java.nio.ByteBuffer. Basically, this is an array of bytes (byte[]) that's wrapped with a suite of methods dedicated to manipulating this array (for example, get(), put(), and so on). A circular buffer (cyclic buffer, ring buffer, or circular queue) is a fixed-size buffer that's connected end-to-end. The following diagram shows us what a circular queue looks like:

Gráfico

Descripción generada automáticamente con confianza media

A circular buffer relies on a pre-allocated array (pre-allocated capacity), but some implementations may require a resizing capability as well. The elements are written/added to the back (*tail*) and removed/read from the front (*head*); this can be seen in the following diagram:

Reloj con fondo blanco y letras negras

Descripción generada automáticamente con confianza media

For the main operations, that is, read (get) and write (put), a circular buffer maintains a pointer (a read pointer and a write pointer). Both pointers are wrapped around the buffer capacity. We can find out how many elements are available to be read and how many free slots can be written whenever we like. This operation takes place in O(1).

A circular byte buffer is a circular buffer of bytes; it can be of chars or some other type. This is exactly what we want to implement here. We can start by writing a stub of our implementation, as follows:

public class CircularByteBuffer {  
  
 private int capacity;  
 private byte[] buffer;  
 private int readPointer;  
 private int writePointer;  
 private int available;  
  
 CircularByteBuffer(int capacity) {  
 this.capacity = capacity;  
 buffer = new byte[capacity];  
 }  
  
 public synchronized int available() {  
 return available;  
 }  
  
 public synchronized int capacity() {  
 return capacity;  
 }  
  
 public synchronized int slots() {  
 return capacity - available;  
 }  
  
 public synchronized void clear() {  
 readPointer = 0;  
 writePointer = 0;  
 available = 0;  
 }  
 ...  
}Copy

Now, let's focus on putting (writing) new bytes and reading (getting) existing bytes. For example, a circular byte buffer that has a capacity of 8 can be represented like so:

Un reloj con números romanos

Descripción generada automáticamente con confianza media

Let's take a look at what's happening at each step:

1. The circular byte buffer is empty and both pointers point to slot 0 (the first slot).
2. We put the 5 bytes corresponding to **hello** in the buffer. readPointer remains in the same position, while writePointer points to slot 5.
3. We get the bytes corresponding with the **h**, so readPointer moves to slot 1.
4. Finally, we attempt to put the bytes of **world** in the buffer. This word is made up of 5 bytes, but we have only four free slots until we reach the buffer capacity. This means we can only write the bytes that correspond with **world**.

Now, let's take a look at the scenario in the following diagram:

Un reloj con números romanos

Descripción generada automáticamente con confianza media

From left to right, the steps are as follows::

1. The first two steps are the same as the ones from the previous scenario.
2. We get the bytes for **hell**. This will move readPointer to position 4.
3. Finally, we put the bytes of **world** in the buffer. This time, the word fits in the buffer and writePointer moves to slot 2.

Based on this flow, we can easily implement a method that puts one byte in the buffer and another that gets one byte from the buffer, as follows:

public synchronized boolean put(int value) {  
 if (available == capacity) {  
 return false;  
 }  
  
 buffer[writePointer] = (byte) value;  
 writePointer = (writePointer + 1) % capacity;  
 available++;  
  
 return true;  
}  
  
public synchronized int get() {  
 if (available == 0) {  
 return -1;  
 }  
  
 byte value = buffer[readPointer];  
 readPointer = (readPointer + 1) % capacity;  
 available--;  
  
 return value;  
}Copy

If we check the Java NIO.2 ByteBuffer API, we'll notice that it exposes several flavors of the get() and put() methods. For example, we should be able to pass a byte[] to the get() method and this method should copy a range of elements from the buffer into this byte[]. The elements are read from the buffer, starting with the current readPointer, and are written in the given byte[], starting from the specified offset.

The following diagram exposes a case where writePointer is greater than readPointer:

Un reloj con números romanos

Descripción generada automáticamente con confianza media

On the left-hand side, we are reading 3 bytes. This moves readPointer from its initial slot, 1, to slot 4. On the right-hand side, we are reading 4 (or more than 4) bytes. Since there are only 4 bytes available, readPointer is moved from its initial slot to the same slot as writePointer (slot 5).

Now, let's analyze a case where writePointer is less than readPointer:

Un reloj con números romanos

Descripción generada automáticamente con confianza media

On the left-hand side, we are reading 3 bytes. This moves readPointer from its initial slot, 6, to slot 1. On the right-hand side, we are reading 4 (or more than 4) bytes. This moves readPointer from its initial slot, 6, to slot 2 (the same slot as writePointer).

Now that we have these two use cases in mind, we can write a get() method in order to copy a range of bytes from the buffer into the given byte[], as follows (this method attempts to read len bytes from the buffer and write them into the given byte[], starting from the given offset):

public synchronized int get(byte[] dest, int offset, int len) {  
  
 if (available == 0) {  
 return 0;  
 }  
  
 int maxPointer = capacity;  
  
 if (readPointer < writePointer) {  
 maxPointer = writePointer;  
 }  
  
 int countBytes = Math.min(maxPointer - readPointer, len);  
 System.arraycopy(buffer, readPointer, dest, offset, countBytes);  
 readPointer = readPointer + countBytes;  
  
 if (readPointer == capacity) {  
 int remainingBytes = Math.min(len - countBytes, writePointer);  
  
 if (remainingBytes > 0) {  
 System.arraycopy(buffer, 0, dest,  
 offset + countBytes, remainingBytes);  
 readPointer = remainingBytes;  
 countBytes = countBytes + remainingBytes;  
 } else {  
 readPointer = 0;  
 }  
 }  
  
 available = available - countBytes;  
  
 return countBytes;  
}Copy

Now, let's focus on putting the given byte[] into the buffer. The elements are read from the given byte[] starting from the specified offset and are written into the buffer starting from the current writePointer. The following diagram exposes a case where writePointer is greater than readPointer:

Un reloj con números romanos

Descripción generada automáticamente

On the left-hand side, we have the initial state of the buffer. So, readPointer points to slot 2, and writePointer points to slot 5. After writing 4 bytes (on the right-hand side), we can see that readPointer was not affected and that writePointer points to slot 1.

The other use case assumes that readPointer is greater than writePointer:

Un reloj con números romanos

Descripción generada automáticamente

On the left-hand side, we have the initial state of the buffer. So, readPointer points to slot 4, and writePointer points to slot 2. After writing 4 bytes (on the right-hand side), we can see that readPointer was not affected and that writePointer points to slot 4. Notice that only two bytes were successfully written. This has happened because we reached the maximum capacity of the buffer before writing all 4 bytes.

Now that we have these two use cases in mind, we can write a put() method in order to copy a range of bytes from the given byte[] into the buffer, as follows (the method attempts to read len bytes from the given byte[] starting from the given offset and attempts to write them into the buffer starting from the current writePointer):

public synchronized int put(byte[] source, int offset, int len) {  
  
 if (available == capacity) {  
 return 0;  
 }  
  
 int maxPointer = capacity;  
  
 if (writePointer < readPointer) {  
 maxPointer = readPointer;  
 }  
  
 int countBytes = Math.min(maxPointer - writePointer, len);  
 System.arraycopy(source, offset, buffer, writePointer, countBytes);  
 writePointer = writePointer + countBytes;  
  
 if (writePointer == capacity) {  
 int remainingBytes = Math.min(len - countBytes, readPointer);  
  
 if (remainingBytes > 0) {  
 System.arraycopy(source, offset + countBytes,  
 buffer, 0, remainingBytes);  
 writePointer = remainingBytes;  
 countBytes = countBytes + remainingBytes;  
 } else {  
 writePointer = 0;  
 }  
 }  
  
 available = available + countBytes;  
  
 return countBytes;  
}Copy

As we mentioned earlier, sometimes, we need to resize the buffer. For example, we may want to double its size by simply calling the resize() method. Basically, this means copying all the available bytes (elements) into a new buffer with double capacity:

public synchronized void resize() {  
  
 byte[] newBuffer = new byte[capacity \* 2];  
  
 if (readPointer < writePointer) {  
 System.arraycopy(buffer, readPointer, newBuffer, 0, available);  
 } else {  
 int bytesToCopy = capacity - readPointer;  
 System.arraycopy(buffer, readPointer, newBuffer, 0, bytesToCopy);  
 System.arraycopy(buffer, 0, newBuffer, bytesToCopy, writePointer);  
 }  
  
 buffer = newBuffer;  
 capacity = buffer.length;  
 readPointer = 0;  
 writePointer = available;  
}Copy

Check the source code bundled with this book to see how it works in full.

146. Tokenizing files

The content in a file is not always received in a way that means it can be processed immediately and will require some additional steps so that it can be prepared for processing. Typically, we need to tokenize the file and extract information from different data structures (arrays, lists, maps, and so on).

For example, let's consider a file, clothes.txt:

Path path = Paths.get("clothes.txt");Copy

Its content is as follows:

Top|white\10/XXL&amp;Swimsuit|black\5/L  
Coat|red\11/M&amp;Golden Jacket|yellow\12/XLDenim|Blue\22/MCopy

This file contains some clothing articles and their details separated by the &amp; character. A single article is represented as follows:

article name | color \ no. available items / sizeCopy

Here, we have several delimiters (&amp;, |, \, /) and a very specific format.

Now, let's take a look at several solutions for extracting and tokenizing the information from this file as a List. We'll collect this information in a utility class, FileTokenizer.

One solution for fetching the articles in a List relies on the String.split() method. Basically, we have to read the file line by line and apply String.split() to each line. The result of tokenizing each line is collected in a List via the List.addAll() method:

public static List<String> get(Path path,   
 Charset cs, String delimiter) throws IOException {  
  
 String delimiterStr = Pattern.quote(delimiter);  
 List<String> content = new ArrayList<>();  
  
 try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
  
 String line;  
 while ((line = br.readLine()) != null) {  
 String[] values = line.split(delimiterStr);  
 content.addAll(Arrays.asList(values));  
 }  
 }  
  
 return content;  
}Copy

Calling this method with the &amp; delimiter will produce the following output:

[Top|white\10/XXL, Swimsuit|black\5/L, Coat|red\11/M, Golden Jacket|yellow\12/XL, Denim|Blue\22/M]Copy

Another flavor of the preceding solution can rely on Collectors.toList() instead of Arrays.asList():

public static List<String> get(Path path,   
 Charset cs, String delimiter) throws IOException {  
  
 String delimiterStr = Pattern.quote(delimiter);  
 List<String> content = new ArrayList<>();  
  
 try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
  
 String line;  
 while ((line = br.readLine()) != null) {  
 content.addAll(Stream.of(line.split(delimiterStr))  
 .collect(Collectors.toList()));  
 }  
 }  
  
 return content;  
}Copy

Alternatively, we can process the content in a lazy manner via Files.lines():

public static List<String> get(Path path,   
 Charset cs, String delimiter) throws IOException {  
  
 try (Stream<String> lines = Files.lines(path, cs)) {  
  
 return lines.map(l -> l.split(Pattern.quote(delimiter)))  
 .flatMap(Arrays::stream)  
 .collect(Collectors.toList());  
 }  
}Copy

For relatively small files, we can load it in memory and process it accordingly:

Files.readAllLines(path, cs).stream()  
 .map(l -> l.split(Pattern.quote(delimiter)))  
 .flatMap(Arrays::stream)  
 .collect(Collectors.toList());Copy

Another solution can rely on JDK 8's Pattern.splitAsStream() method. This method creates a stream from the given input sequence. For the sake of variation, this time, let's collect the resulted list via Collectors.joining(";"):

public static List<String> get(Path path,   
 Charset cs, String delimiter) throws IOException {  
  
 Pattern pattern = Pattern.compile(Pattern.quote(delimiter));  
 List<String> content = new ArrayList<>();  
  
 try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
  
 String line;  
 while ((line = br.readLine()) != null) {  
 content.add(pattern.splitAsStream(line)  
 .collect(Collectors.joining(";")));  
 }  
 }  
 return content;  
}Copy

Let's call this method with the &amp; delimiter:

List<String> tokens = FileTokenizer.get(  
 path, StandardCharsets.UTF\_8, "&amp;");Copy

The result is as follows:

[Top|white\10/XXL;Swimsuit|black\5/L, Coat|red\11/M;Golden Jacket|yellow\12/XL, Denim|Blue\22/M]Copy

So far, the presented solutions obtain a list of articles by applying a single delimiter. But sometimes, we need to apply more delimiters. For example, let's assume that we want to obtain the following output (list):

[Top, white, 10, XXL, Swimsuit, black, 5, L, Coat, red, 11, M, Golden Jacket, yellow, 12, XL, Denim, Blue, 22, M]Copy

In order to obtain this list, we have to apply several delimiters (&amp;, |, \, and /). This can be accomplished by using String.split() and passing a regular expression based on the logical OR operator (*x*|*y*) to it:

public static List<String> getWithMultipleDelimiters(  
 Path path, Charset cs, String...delimiters) throws IOException {  
  
 String[] escapedDelimiters = new String[delimiters.length];  
 Arrays.setAll(escapedDelimiters, t -> Pattern.quote(delimiters[t]));  
 String delimiterStr = String.join("|", escapedDelimiters);  
  
 List<String> content = new ArrayList<>();  
  
 try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
  
 String line;  
 while ((line = br.readLine()) != null) {  
 String[] values = line.split(delimiterStr);  
 content.addAll(Arrays.asList(values));  
 }  
 }  
  
 return content;  
}Copy

Let's call this method with our delimiters (&amp;, |, \, and /) to obtain the required result:

List<String> tokens = FileTokenizer.getWithMultipleDelimiters(  
 path, StandardCharsets.UTF\_8,   
 new String[] {"&amp;", "|", "\\", "/"});Copy

Ok; so far, so good! All of these solutions are based on String.split() and Pattern.splitAsStream(). Another set of solutions can rely on the StringTokenizer class (it doesn't excel at performance, so use it carefully). This class can apply a delimiter (or more than one) to the given string and expose the two main methods for controlling it, that is, hasMoreElements() and nextToken():

public static List<String> get(Path path,  
 Charset cs, String delimiter) throws IOException {  
  
 StringTokenizer st;  
 List<String> content = new ArrayList<>();  
  
 try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
  
 String line;  
 while ((line = br.readLine()) != null) {  
 st = new StringTokenizer(line, delimiter);  
 while (st.hasMoreElements()) {  
 content.add(st.nextToken());  
 }  
 }  
 }  
  
 return content;  
}Copy

It can be used in conjunction with Collectors as well:

public static List<String> get(Path path,   
 Charset cs, String delimiter) throws IOException {  
  
 List<String> content = new ArrayList<>();  
  
 try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
  
 String line;  
 while ((line = br.readLine()) != null) {  
 content.addAll(Collections.list(  
 new StringTokenizer(line, delimiter)).stream()  
 .map(t -> (String) t)  
 .collect(Collectors.toList()));  
 }  
 }  
  
 return content;  
}Copy

Multiple delimiters can be used if we separate them using //:

public static List<String> getWithMultipleDelimiters(  
 Path path, Charset cs, String...delimiters) throws IOException {  
  
 String delimiterStr = String.join("//", delimiters);  
 StringTokenizer st;  
 List<String> content = new ArrayList<>();  
  
 try (BufferedReader br = Files.newBufferedReader(path, cs)) {  
  
 String line;  
 while ((line = br.readLine()) != null) {  
 st = new StringTokenizer(line, delimiterStr);  
 while (st.hasMoreElements()) {  
 content.add(st.nextToken());  
 }  
 }  
 }  
  
 return content;  
}Copy

For better performance and regular expression support (that is, high flexibility) it is advisable to rely on String.split() instead of StringTokenizer. From the same category, consider the *Working with Scanner* section as well.

147. Writing formatted output directly to a file

Let's suppose that we have 10 numbers (integers and doubles) and we want them to be nicely formatted (have an indentation, alignment, and a number of decimals that sustain readability and usefulness) in a file.

In our first attempt, we wrote them to the file like so (no formatting was applied):

Path path = Paths.get("noformatter.txt");  
  
try (BufferedWriter bw = Files.newBufferedWriter(path,  
 StandardCharsets.UTF\_8, StandardOpenOption.CREATE,  
 StandardOpenOption.WRITE)) {  
  
 for (int i = 0; i < 10; i++) {  
 bw.write("| " + intValues[i] + " | " + doubleValues[i] + " | ");  
 bw.newLine();  
 }  
}Copy

The output of the preceding code is similar to what's shown on the left-hand side of the following diagram:

Texto

Descripción generada automáticamente

However, we want to obtain the result that's shown on the right-hand side of the preceding diagram. In order to solve this problem, we need to use the String.format() method. This method allows us to specify the format rules as a string that respects the following pattern:

%[*flags*][*width*][.*precision*]*conversion-character*Copy

Now, let's take a look at what represents each component of this pattern:

* [flags] is optional and consists of standard approaches for modifying the output. Often, they are used for formatting integers and floating-point numbers.
* [width] is optional and sets the field width for our output (the minimum number of characters written to the output).
* [.precision] is optional and specifies the number of digits of precision for floating-point values (or the length of a substring to extract from a String).
* conversion-character is mandatory and tells us how the argument will be formatted. The most used conversion-characters are as follows:
  + s: Used for formatting strings
  + d: Used for formatting decimal integers
  + f: Used for formatting floating-point numbers
  + t: Used for formatting date/time values

As a line separator, we can use %n.

With this knowledge of formatting rules, we can obtain what we want as follows (%6s is used for the integers while %.3f is used for the doubles):

Path path = Paths.get("withformatter.txt");  
  
try (BufferedWriter bw = Files.newBufferedWriter(path,  
 StandardCharsets.UTF\_8, StandardOpenOption.CREATE,   
 StandardOpenOption.WRITE)) {  
  
 for (int i = 0; i<10; i++) {  
 bw.write(String.format("| %6s | %.3f |",  
 intValues[i], doubleValues[i]));  
 bw.newLine();  
 }  
}Copy

Another solution can be provided via the Formatter class. This class is dedicated to format strings and uses the same formatting rules as String.format(). It has a format() method, which we can use to rewrite the preceding snippet of code:

Path path = Paths.get("withformatter.txt");  
  
try (Formatter output = new Formatter(path.toFile())) {  
  
 for (int i = 0; i < 10; i++) {  
 output.format("| %6s | %.3f |%n", intValues[i], doubleValues[i]);  
 }  
}Copy

How about formatting only the integer's numbers?

Texto

Descripción generada automáticamente

Well, we can obtain this by applying a DecimalFormat and a string formatter, as follows:

Path path = Paths.get("withformatter.txt");  
DecimalFormat formatter = new DecimalFormat("###,### bytes");  
  
try (Formatter output = new Formatter(path.toFile())) {  
  
 for (int i = 0; i < 10; i++) {  
 output.format("%12s%n", formatter.format(intValues[i]));  
 }  
}Copy

148. Working with Scanner

Scanner exposes an API for parsing text from strings, files, the console, and so on. Parsing is the process of tokenizing the given input and returning it as needed (for example, integers, floats, doubles, and so on). By default, Scanner parses the given input by using a white space (default delimiter) and exposes the tokens via a suite of next*Foo*() methods (for example, next(), nextLine(), nextInt(), nextDouble(), and so on).

From the same category of problems, consider the *Tokenizing files* section as well.

For example, let's assume that we have a file (doubles.txt) that contains double numbers separated by spaces, as shown in the following illustration:

Texto

Descripción generada automáticamente

If we want to obtain this text as doubles, then we can read it and rely on a snippet of *spaghetti* code to tokenize and convert it into doubles. Alternatively, we can rely on Scanner and its nextDouble() method, as follows:

try (Scanner scanDoubles = new Scanner(  
 Path.of("doubles.txt"), StandardCharsets.UTF\_8)) {  
  
 while (scanDoubles.hasNextDouble()) {  
 double number = scanDoubles.nextDouble();  
 System.out.println(number);  
 }  
}Copy

The output of the preceding code is as follows:

23.4556  
1.23  
...Copy

However, a file may contain mixed information of different types. For example, the file (people.txt) in the following illustration contains strings and integers that are separated by different delimiters (a comma and a semicolon):

Texto

Descripción generada automáticamente

Scanner exposes a method called useDelimiter(). This method takes an argument of the String or Pattern type in order to specify the delimiter(s) that should be used as a regular expression:

try (Scanner scanPeople = new Scanner(Path.of("people.txt"),  
 StandardCharsets.UTF\_8).useDelimiter(";|,")) {  
  
 while (scanPeople.hasNextLine()) {  
 System.out.println("Name: " + scanPeople.next().trim());  
 System.out.println("Surname: " + scanPeople.next());  
 System.out.println("Age: " + scanPeople.nextInt());  
 System.out.println("City: " + scanPeople.next());  
 }  
}Copy

The output of using this method is as follows:

Name: Matt  
Surname: Kyle  
Age: 23  
City: San Francisco  
...Copy

Starting with JDK 9, Scanner exposes a new method called tokens(). This method returns a stream of delimiter-separated tokens from Scanner. For example, we can use it to parse the people.txt file and print it on the console, as follows:

try (Scanner scanPeople = new Scanner(Path.of("people.txt"),  
 StandardCharsets.UTF\_8).useDelimiter(";|,")) {  
  
 scanPeople.tokens().forEach(t -> System.out.println(t.trim()));  
}Copy

The output of using the preceding method is as follows:

Matt  
Kyle  
23  
San Francisco  
...Copy

Alternatively, we can join the tokens by space:

try (Scanner scanPeople = new Scanner(Path.of("people.txt"),  
 StandardCharsets.UTF\_8).useDelimiter(";|,")) {  
  
 String result = scanPeople.tokens()  
 .map(t -> t.trim())  
 .collect(Collectors.joining(" "));  
}Copy

In the *Searching in big files* section, there is an example of how to use this method to search for a certain piece of text in a file.

The output of using the preceding method is as follows:

Matt Kyle 23 San Francisco Darel Der 50 New York ...Copy

In terms of the tokens() methods, JDK 9 also comes with a method called findAll(). This is a very handy method for finding all the tokens that respect a certain regular expression (provided as a String or Pattern). This method returns a Stream<MatchResult> and can be used like so:

try (Scanner sc = new Scanner(Path.of("people.txt"))) {  
  
 Pattern pattern = Pattern.compile("4[0-9]");  
  
 List<String> ages = sc.findAll(pattern)  
 .map(MatchResult::group)  
 .collect(Collectors.toList());  
  
 System.out.println("Ages: " + ages);  
}Copy

The preceding code selects all the tokens that represent ages between 40 and 49 years old, that is, 40, 43, and 43.

Scanner is a convenient approach to use if we wish to parse the input that's provided in the console:

Scanner scanConsole = new Scanner(System.in);  
  
String name = scanConsole.nextLine();  
String surname = scanConsole.nextLine();  
int age = scanConsole.nextInt();  
// an int cannot include "\n" so we need  
//the next line just to consume the "\n"  
scanConsole.nextLine();  
String city = scanConsole.nextLine();Copy

Note that, for numeric inputs (read via nextInt(), nextFloat(), and so on), we need to consume the newline character as well (this occurs when we hit *Enter*). Basically, Scanner will not fetch this character when parsing a number, and so it will go in the next token. If we don't consume it by adding a nextLine() code line then, from this point forward, the inputs will become unaligned and lead to an exception of the InputMismatchException type or come to a premature end.  
The Scanner constructors that support charsets were introduced in JDK 10.

Let's take a look at the difference between Scanner and BufferedReader.

Scanner versus BufferedReader

So, should we use Scanner or BufferedReader? Well, if we need to parse the file, then Scanner is the way to go; otherwise, BufferedReader is more suitable. A head-to-head comparison of them will reveal the following:

* BufferedReader is faster than Scanner since it doesn't perform any parsing operations.
* BufferedReader excels when it comes to reading while Scanner excels when it comes to parsing.
* By default, BufferedReader uses a buffer of 8 KB, while Scanner uses a buffer of 1 KB.
* BufferedReader is a good fit for reading long strings, while Scanner is better for short inputs.
* BufferedReader is synchronized, but Scanner is not.
* A Scanner can use a BufferedReader, while the opposite is not possible. This is shown in the following code:

try (Scanner scanDoubles = new Scanner(Files.newBufferedReader(  
 Path.of("doubles.txt"), StandardCharsets.UTF\_8))) {   
 ...  
}Copy

Summary

We've reached the end of this chapter, where we covered various I/O-specific problems. From manipulating, walking, and watching paths to streaming files and efficient ways of reading/writing text and binary files, we have covered a lot.

Download the applications from this chapter to view the results and additional details.

# Java Reflection Classes, Interfaces, Constructors, Methods, and Fields

This chapter includes 17 problems that involve the Java Reflection API. From classical topics such as inspecting and instantiating Java artifacts (for example, modules, packages, classes, interfaces, superclasses, constructors, methods, annotations, and arrays) to *synthetic* and *bridge* constructs or nest-based access control (JDK 11), this chapter provides solid coverage of the Java Reflection API. By the end of this chapter, the Java Reflection API will have no secrets left unturned, and you will be ready to show your colleagues what reflection can do.

Problems

Use the following problems to test your Java Reflection API programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Inspecting packages:**Write several examples for inspecting Java packages (for example, names, a list of classes, and so on).
2. **Inspecting classes and superclasses:**Write several examples for inspecting classes and superclasses (for example, get Class via the class name, modifiers, implemented interfaces, constructors, methods, and fields).
3. **Instantiating via a reflected constructor:**Write a program that creates instances via reflection.
4. **Getting the annotation of a receiver type:**Write a program that gets the annotation on a receiver type.
5. **Getting synthetic and bridge constructs:**Write a program that gets *synthetic* and *bridge* constructs via reflection.
6. **Checking the variable number of arguments:**Write a program that checks whether a method gets a variable number of arguments.
7. **Checking default methods:**Write a program that checks whether a method is default.
8. **Nest-based access control via reflection:**Write a program that provides access to nest-based constructs via reflection.
9. **Reflection for getters and setters:**Write several examples that invoke getters and setters via reflection. Additionally, write a program that generates getters and setters via reflection.
10. **Reflecting annotations:**Write several examples of fetching different kinds of annotations via reflection.
11. **Invoking an instance method:**Write a program that invokes an instance method via reflection.
12. **Getting static methods:**Write a program that groups the static methods of the given class and invokes one of them via reflection.
13. **Getting generic types of methods, fields, and exceptions:**Write a program that fetches the generic types of the given methods, fields, and exceptions via reflection.
14. **Getting public and private fields:**Write a program that fetches the public and private fields of the given class via reflection.
15. **Working with arrays:**Write several examples for working with arrays via reflection.
16. **Inspecting modules:**Write several examples for inspecting Java 9 modules via reflection.
17. **Dynamic proxies:**Write a program that relies on *dynamic proxies* for counting the number of invocations of the methods of the given interfaces.

Solutions

The following sections describe the solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations that are shown here only include the most interesting and important details that are needed to solve the problems. You can download the example solutions to view additional details and experiment with the programs from <https://github.com/PacktPublishing/Java-Coding-Problems>.

149. Inspecting packages

The java.lang.Package class is our main focus when we need to obtain information about a specific package. Using this class, we can find out the package's name, the vendor that implemented this package, its title, the version of the package, and so on.

This class is commonly used to find the name of a package that contains a certain class. For example, the package name of the Integer class can be easily obtained as follows:

Class clazz = Class.forName("java.lang.Integer");  
Package packageOfClazz = clazz.getPackage();  
  
// java.lang  
String packageNameOfClazz = packageOfClazz.getName();Copy

Now, let's find the package name of the File class:

File file = new File(".");  
Package packageOfFile = file.getClass().getPackage();  
  
// java.io  
String packageNameOfFile = packageOfFile.getName();Copy

If we are trying to find the package name of the current class, then we can rely on this.getClass().getPackage().getName(). This works in a non-static context.

But if all we want is to quickly list all the packages of the current class loader, then we can rely on the getPackages() method, as follows:

Package[] packages = Package.getPackages();Copy

Based on the getPackages() method, we can list all the packages defined by the caller's class loader, as well as its ancestors, which start with a given prefix, as follows:

public static List<String> fetchPackagesByPrefix(String prefix) {  
  
 return Arrays.stream(Package.getPackages())  
 .map(Package::getName)  
 .filter(n -> n.startsWith(prefix))  
 .collect(Collectors.toList());  
}Copy

If this method lives in a utility class named Packages, then we can call it as follows:

List<String> packagesSamePrefix   
 = Packages.fetchPackagesByPrefix("java.util");Copy

You will see output similar to the following:

java.util.function, java.util.jar, java.util.concurrent.locks,  
java.util.spi, java.util.logging, ...Copy

Sometimes, we just want to list all the classes of a package in the system class loader. Let's see how we can do this.

Getting the classes of a package

For example, we may want to list the classes from one of the packages of the current application (for example, the modern.challenge package) or the classes from one of the packages from our compile-time libraries (for example, commons-lang-2.4.jar).

Classes are wrapped in packages that can be archived in JARs, though they don't have to be. In order to cover both cases, we need to discover whether the given package lives in a JAR or not. We can do this by loading the resource via ClassLoader.getSystemClassLoader().getResource(package\_path) and checking the returned URL of the resource. If the package doesn't live in a JAR, then a resource will be a URL starting with the file: scheme, as in the following example (we are using modern.challenge):

file:/D:/Java%20Modern%20Challenge/Code/Chapter%207/Inspect%20packages/build/classes/modern/challengeCopy

But if the package is inside a JAR (for example, org.apache.commons.lang3.builder), then the URL will start with the jar: scheme, as in the following example:

jar:file:/D:/.../commons-lang3-3.9.jar!/org/apache/commons/lang3/builderCopy

If we take into consideration that a resource of a package from a JAR starts with the jar: prefix, then we can write a method to distinguish between them, as follows:

private static final String JAR\_PREFIX = "jar:";  
  
public static List<Class<?>> fetchClassesFromPackage(  
 String packageName) throws URISyntaxException, IOException {  
  
 List<Class<?>> classes = new ArrayList<>();  
 String packagePath = packageName.replace('.', '/');  
  
 URL resource = ClassLoader  
 .getSystemClassLoader().getResource(packagePath);  
  
 if (resource != null) {  
 if (resource.toString().startsWith(JAR\_PREFIX)) {  
 classes.addAll(fetchClassesFromJar(resource, packageName));  
 } else {  
 File file = new File(resource.toURI());  
 classes.addAll(fetchClassesFromDirectory(file, packageName));  
 }  
 } else {  
 throw new RuntimeException("Resource not found for package: "   
 + packageName);  
 }  
  
 return classes;  
}Copy

So, if the given package is in a JAR, then we call another helper method, fetchClassesFromJar(); otherwise, we call this helper method, fetchClassesFromDirectory(). As their names suggest, these helpers know how to extract the classes of the given package from a JAR or from a directory.

Mainly, these two methods are just some snippets of *spaghetti* code that are meant to identify the files that have the .class extension. Each class is passed through Class.forName() to ensure that it is returned as Class, not as String. Both methods are available in the code bundled with this book.

How about listing the classes from packages that are not in the system class loader, for example, a package from an external JAR? A convenient way to accomplish this relies on URLClassLoader. This class is used to load classes and resources from a search path of URLs that refer to both JAR files and directories. We will deal only with JARs, but it is pretty straightforward to do this for directories as well.

So, based on the given path, we need to fetch all the JARs and return them as URL[] (this array is needed to define URLClassLoader). For example, we can rely on the Files.find() method to traverse the given path and extract all the JARs, like so:

public static URL[] fetchJarsUrlsFromClasspath(Path classpath)  
 throws IOException {  
  
 List<URL> urlsOfJars = new ArrayList<>();  
 List<File> jarFiles = Files.find(  
 classpath,  
 Integer.MAX\_VALUE,  
 (path, attr) -> !attr.isDirectory() &&  
 path.toString().toLowerCase().endsWith(JAR\_EXTENSION))  
 .map(Path::toFile)  
 .collect(Collectors.toList());  
  
 for (File jarFile: jarFiles) {  
  
 try {  
 urlsOfJars.add(jarFile.toURI().toURL());  
 } catch (MalformedURLException e) {  
 logger.log(Level.SEVERE, "Bad URL for{0} {1}",  
 new Object[] {  
 jarFile, e  
 });  
 }  
 }  
  
 return urlsOfJars.toArray(URL[]::new);  
}Copy

Notice that we are scanning all the subdirectories, starting with the given path. Of course, this is a design decision and it is easy to parameterize the depth of searching. For now, let's fetch the JARs from the tomcat8/lib folder (there is no need to install Tomcat especially for this; just use any other local directory of JARs and do the proper modifications):

URL[] urls = Packages.fetchJarsUrlsFromClasspath(  
 Path.of("D:/tomcat8/lib"));Copy

Now, we can instantiate URLClassLoader:

URLClassLoader urlClassLoader = new URLClassLoader(  
 urls, Thread.currentThread().getContextClassLoader());Copy

This will construct a new URLClassLoader object for the given URLs and will use the current class loader for delegation (the second argument can be null as well). Our URL[] points only to JARs, but as a rule of thumb, any jar: scheme URL is assumed to refer to a JAR file, and any file: scheme URL that ends with / is assumed to refer to a directory.

One of the JARs that's present in the tomcat8/lib folder is called tomcat-jdbc.jar. In this JAR, there is a package called org.apache.tomcat.jdbc.pool. Let's list the classes of this package:

List<Class<?>> classes = Packages.fetchClassesFromPackage(  
 "org.apache.tomcat.jdbc.pool", urlClassLoader);Copy

The fetchClassesFromPackage() method is a helper that simply scans the URL[] array of URLClassLoader and fetches the classes that are in the given package. Its source code is available with the code bundled with this book.

Inspecting packages inside modules

If we go with Java 9 modularity, then our packages will live inside modules. For example, if we have a class called Manager in a package called com.management in a module called org.tournament, then we can fetch all the packages of this module like so:

Manager mgt = new Manager();  
Set<String> packages = mgt.getClass().getModule().getPackages();Copy

In addition, if we want to create a class, then we need the following Class.forName() flavor:

Class<?> clazz = Class.forName(mgt.getClass()  
 .getModule(), "com.management.Manager");Copy

Keep in mind that each module is represented on disk as a directory with the same name. For example, the org.tournament module is on disk a folder with this name. Moreover, each module is mapped as a separate JAR with this name (for example, org.tournament.jar). By having these coordinates in mind, it is pretty straightforward to adapt the code from this section so that it lists all the classes of a given package of a given module.

150. Inspecting classes

By using the Java Reflection API, we can examine the details of a class—an object's class name, modifiers, constructors, methods, fields, implemented interfaces, and so on.

Let's assume that we have the following Pair class:

public final class Pair<L, R> extends Tuple implements Comparable {  
  
 final L left;  
 final R right;  
  
 public Pair(L left, R right) {  
 this.left = left;  
 this.right = right;  
 }  
  
 public class Entry<L, R> {}  
 ...  
}Copy

Let's also assume that we have an instance of it:

Pair pair = new Pair(1, 1);Copy

Now, let's use reflection to get the name of the Pair class.

Get the name of the Pair class via an instance

By having an instance (an object) of Pair, we can find out the name of its class by calling the getClass() method, as well as Class.getName(), getSimpleName(), and getCanonicalName(), as shown in the following example:

Class<?> clazz = pair.getClass();  
  
// modern.challenge.Pair  
System.out.println("Name: " + clazz.getName());  
  
// Pair  
System.out.println("Simple name: " + clazz.getSimpleName());  
  
// modern.challenge.Pair  
System.out.println("Canonical name: " + clazz.getCanonicalName());Copy

An anonymous class doesn't have simple and canonical names.

Notice that getSimpleName() returns the unqualified class name. Alternatively, we can obtain the class as follows:

Class<Pair> clazz = Pair.class;  
Class<?> clazz = Class.forName("modern.challenge.Pair");Copy

Getting the Pair class modifiers

In order to get the modifiers (public, protected, private, final, static, abstract, and interface) of a class, we can call the Class.getModifiers() method. This method returns an int value representing each modifier as a flag bit. For decoding the result, we rely on the Modifier class, as follows:

int modifiers = clazz.getModifiers();  
  
System.out.println("Is public? "   
 + Modifier.isPublic(modifiers)); // true  
System.out.println("Is final? "   
 + Modifier.isFinal(modifiers)); // true  
System.out.println("Is abstract? "   
 + Modifier.isAbstract(modifiers)); // falseCopy

Getting the Pair class implemented interfaces

In order to obtain the interfaces that are directly implemented by a class or interface represented by an object, we simply call Class.getInterfaces(). This method returns an array. Since the Pair class implements a single interface (Comparable), the returned array will contain a single element:

Class<?>[] interfaces = clazz.getInterfaces();  
  
// interface java.lang.Comparable  
System.out.println("Interfaces: " + Arrays.toString(interfaces));  
  
// Comparable  
System.out.println("Interface simple name: "   
 + interfaces[0].getSimpleName());Copy

Getting the Pair class constructors

The public constructors of a class can be obtained via the Class.getConstructors() class. The returned result is Constructor<?>[]:

Constructor<?>[] constructors = clazz.getConstructors();  
  
// public modern.challenge.Pair(java.lang.Object,java.lang.Object)  
System.out.println("Constructors: " + Arrays.toString(constructors));Copy

For fetching all the declared constructors (for example, private and protected constructors), call getDeclaredConstructors(). When searching for a certain constructor, call getConstructor​(Class<?>... parameterTypes) or getDeclaredConstructor​(Class<?>... parameterTypes).

Getting the Pair class fields

All the fields of a class are accessible via the Class.getDeclaredFields() method. This method returns an array of Field:

Field[] fields = clazz.getDeclaredFields();  
  
// final java.lang.Object modern.challenge.Pair.left  
// final java.lang.Object modern.challenge.Pair.right  
System.out.println("Fields: " + Arrays.toString(fields));Copy

For fetching the actual name of the fields, we can easily provide a helper method:

public static List<String> getFieldNames(Field[] fields) {  
  
 return Arrays.stream(fields)  
 .map(Field::getName)  
 .collect(Collectors.toList());  
}Copy

Now, we only receive the names of the fields:

List<String> fieldsName = getFieldNames(fields);  
  
// left, right  
System.out.println("Fields names: " + fieldsName);Copy

Getting the value of a field can be done via a general method named Object get(Object obj) and via a set of get*Foo*() methods (consider documentation for details). The obj represents a static or instance field. For example, let's assume the ProcedureOutputs class that have has a private field named callableStatement which is of type CallableStatement. Let's use Field.get() method to access this field for checking if the CallableStatement is closed:

ProcedureOutputs procedureOutputs   
 = storedProcedure.unwrap(ProcedureOutputs.class);  
  
Field csField = procedureOutputs.getClass()  
 .getDeclaredField("callableStatement");   
csField.setAccessible(true);  
  
CallableStatement cs   
 = (CallableStatement) csField.get(procedureOutputs);  
  
System.out.println("Is closed? " + cs.isClosed());Copy

For fetching only the public fields, call getFields(). For searching for a certain field, call getField​(String fieldName) or getDeclaredField​(String name).

Getting the Pair class methods

The public methods of a class are accessible via the Class.getMethods() method. This method returns an array of Method:

Method[] methods = clazz.getMethods();  
// public boolean modern.challenge.Pair.equals(java.lang.Object)  
// public int modern.challenge.Pair.hashCode()  
// public int modern.challenge.Pair.compareTo(java.lang.Object)  
// ...  
System.out.println("Methods: " + Arrays.toString(methods));Copy

For fetching the actual name of the methods, we can quickly provide a helper method:

public static List<String> getMethodNames(Method[] methods) {  
  
 return Arrays.stream(methods)  
 .map(Method::getName)  
 .collect(Collectors.toList());  
}Copy

Now, we only retrieve the names of the methods:

List<String> methodsName = getMethodNames(methods);  
  
// equals, hashCode, compareTo, wait, wait,  
// wait, toString, getClass, notify, notifyAll  
System.out.println("Methods names: " + methodsName);Copy

For fetching all the declared methods (for example, private and protected), call getDeclaredMethods(). For searching for a certain method, call getMethod​(String name, Class<?>... parameterTypes) or getDeclaredMethod​(String name, Class<?>... parameterTypes).

Getting the Pair class module

If we go with JDK 9 modularity, then our classes will live inside modules. The Pair class is not in a module, but we can easily get the module of a class via JDK 9's Class.getModule() method (if the class is not in a module, then this method returns null):

// null, since Pair is not in a Module  
Module module = clazz.getModule();Copy

Getting the Pair class superclass

The Pair class extends the Tuple class; therefore, the Tuple class is a superclass of Pair. We can obtain it via the Class.getSuperclass() method, as follows:

Class<?> superClass = clazz.getSuperclass();  
// modern.challenge.Tuple  
System.out.println("Superclass: " + superClass.getName());Copy

Getting the name of a certain type

Starting with JDK 8, we can get an informative string for the name of a certain type.

This method returns the same string as one or more of getName(), getSimpleName(), or getCanonicalName():

* For primitives, it returns the same for all three methods:

System.out.println("Type: " + int.class.getTypeName()); // intCopy

* For Pair, it returns the same thing as getName() and getCanonicalName():

// modern.challenge.Pair  
System.out.println("Type name: " + clazz.getTypeName());Copy

* For inner classes (like Entry is for Pair), it returns the same thing as getName():

// modern.challenge.Pair$Entry  
System.out.println("Type name: "   
 + Pair.Entry.class.getTypeName());Copy

* For an anonymous class, it returns the same thing as getName():

Thread thread = new Thread() {  
 public void run() {  
 System.out.println("Child Thread");  
 }  
};  
  
// modern.challenge.Main$1  
System.out.println("Anonymous class type name: "  
 + thread.getClass().getTypeName());Copy

* For arrays, it returns the same thing as getCanonicalName():

Pair[] pairs = new Pair[10];  
// modern.challenge.Pair[]  
System.out.println("Array type name: "   
 + pairs.getClass().getTypeName());Copy

Getting a string that describes the class

Starting with JDK 8, we can obtain a quick description of a class (containing modifiers, name, types parameters, and so on) via the Class.toGenericString() method.

Let's take a look at several examples:

// public final class modern.challenge.Pair<L,R>  
System.out.println("Description of Pair: "   
 + clazz.toGenericString());  
  
// public abstract interface java.lang.Runnable  
System.out.println("Description of Runnable: "   
 + Runnable.class.toGenericString());  
  
// public abstract interface java.util.Map<K,V>  
System.out.println("Description of Map: "   
 + Map.class.toGenericString());Copy

Getting the type descriptor string for a class

Starting with JDK 12, we can obtain the type descriptor of a class as a String object via the Class.descriptorString() method:

// Lmodern/challenge/Pair;  
System.out.println("Type descriptor of Pair: "   
 + clazz.descriptorString());  
  
// Ljava/lang/String;  
System.out.println("Type descriptor of String: "   
 + String.class.descriptorString());Copy

Getting the component type of an array

For arrays only, JDK 12 provides the Class<?> componentType() method. This method returns the component type of the array, as shown in the following two examples:

Pair[] pairs = new Pair[10];  
String[] strings = new String[] {"1", "2", "3"};  
  
// class modern.challenge.Pair  
System.out.println("Component type of Pair[]: "   
 + pairs.getClass().componentType());  
  
// class java.lang.String  
System.out.println("Component type of String[]: "   
 + strings.getClass().componentType());Copy

Getting a class for an array type whose component type is described by Pair

Starting with JDK 12, we can get Class for an array type whose component type is described by the given class via Class.arrayType():

Class<?> arrayClazz = clazz.arrayType();  
  
// modern.challenge.Pair<L,R>[]  
System.out.println("Array type: " + arrayClazz.toGenericString());Copy

151. Instantiating via a reflected constructor

We can instantiate a class via Constructor.newInstance() using the Java Reflection API.

Let's consider the following class, which has four constructors:

public class Car {  
  
 private int id;  
 private String name;  
 private Color color;  
  
 public Car() {}  
  
 public Car(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public Car(int id, Color color) {  
 this.id = id;  
 this.color = color;  
 }  
  
 public Car(int id, String name, Color color) {  
 this.id = id;  
 this.name = name;  
 this.color = color;  
 }  
  
 // getters and setters omitted for brevity  
}Copy

A Car instance can be created via one of these four constructors. The Constructor class exposes a method that takes the types of the parameters of a constructor and returns a Constructor object that reflects the matched constructor. This method is called getConstructor​(Class<?>... parameterTypes).

Let's call each of the preceding constructors:

Class<Car> clazz = Car.class;  
  
Constructor<Car> emptyCnstr   
 = clazz.getConstructor();  
  
Constructor<Car> idNameCnstr   
 = clazz.getConstructor(int.class, String.class);  
  
Constructor<Car> idColorCnstr   
 = clazz.getConstructor(int.class, Color.class);  
  
Constructor<Car> idNameColorCnstr   
 = clazz.getConstructor(int.class, String.class, Color.class);Copy

Furthermore, Constructor.newInstance​(Object... initargs) can return an instance of Car that corresponds with the invoked constructor:

Car carViaEmptyCnstr = emptyCnstr.newInstance();  
  
Car carViaIdNameCnstr = idNameCnstr.newInstance(1, "Dacia");  
  
Car carViaIdColorCnstr = idColorCnstr  
 .newInstance(1, new Color(0, 0, 0));  
  
Car carViaIdNameColorCnstr = idNameColorCnstr  
 .newInstance(1, "Dacia", new Color(0, 0, 0));Copy

Now, is time to see how we can instantiate a private constructor via reflection.

Instantiating a class via a private constructor

The Java Reflection API can be used to instantiate a class via its private constructor as well. For example, let's suppose that we have a utility class called Cars. Following best practices, we will define this class as final and with a private constructor to not allow instances:

public final class Cars {  
  
 private Cars() {}  
 // static members  
}Copy

Fetching this constructor can be accomplished via Class.getDeclaredConstructor(), as follows:

Class<Cars> carsClass = Cars.class;  
Constructor<Cars> emptyCarsCnstr = carsClass.getDeclaredConstructor();Copy

Calling newInstance() at this instance will throw IllegalAccessException since the invoked constructor has private access. However, Java Reflection allows us to modify the access level via the flag method, Constructor.setAccessible(). This time, the instantiation works as expected:

emptyCarsCnstr.setAccessible(true);  
Cars carsViaEmptyCnstr = emptyCarsCnstr.newInstance();Copy

In order to block this approach, it is advisable to throw an error from a private constructor, as follows:

public final class Cars {  
  
 private Cars() {  
 throw new AssertionError("Cannot be instantiated");  
 }  
  
 // static members  
}Copy

This time, the instantiation attempt will fail with AssertionError.

Instantiating a class from a JAR

Let's suppose that we have the Guava JAR in the D:/Java Modern Challenge/Code/lib/ folder, and we want to create an instance of CountingInputStream and read one byte from a file.

First, we define a URL[] array for the Guava JAR, as follows:

URL[] classLoaderUrls = new URL[] {  
 new URL(  
 "file:///D:/Java Modern Challenge/Code/lib/guava-16.0.1.jar")  
};Copy

Then, we will define URLClassLoader for this URL[] array:

URLClassLoader urlClassLoader = new URLClassLoader(classLoaderUrls);Copy

Next, we will load the target class (CountingInputStream is a class that counts the number of bytes that are read from InputStream):

Class<?> cisClass = urlClassLoader.loadClass(  
 "com.google.common.io.CountingInputStream");Copy

Once the target class has been loaded, we can fetch its constructor (CountingInputStream has a single constructor that wraps the given InputStream):

Constructor<?> constructor   
 = cisClass.getConstructor(InputStream.class);Copy

Furthermore, we can create an instance of CountingInputStream via this constructor:

Object instance = constructor.newInstance(  
 new FileInputStream​(Path.of("test.txt").toFile()));Copy

In order to ensure that the returned instance is operational, let's call two of its methods (the read() method reads a single byte at once, while the getCount() method returns the number of read bytes):

Method readMethod = cisClass.getMethod("read");  
Method countMethod = cisClass.getMethod("getCount");Copy

Next, let's read a single byte and see what getCount() returns:

readMethod.invoke(instance);  
Object readBytes = countMethod.invoke(instance);  
System.out.println("Read bytes (should be 1): " + readBytes); // 1Copy

Useful snippets of code

As a bonus, let's look at several snippets of code that are commonly needed when working with reflection and constructors.

First, let's fetch the number of available constructors:

Class<Car> clazz = Car.class;  
Constructor<?>[] cnstrs = clazz.getConstructors();  
System.out.println("Car class has "   
 + cnstrs.length + " constructors"); // 4Copy

Now, let's see how many parameters have each of these four constructors:

for (Constructor<?> cnstr : cnstrs) {  
 int paramCount = cnstr.getParameterCount();  
 System.out.println("\nConstructor with "   
 + paramCount + " parameters");  
}Copy

In order to get details about each parameter of a constructor, we can call Constructor.getParameters(). This method returns an array of Parameter (this class was added in JDK 8 and it provides a comprehensive list of methods for *dissecting* a parameter):

for (Constructor<?> cnstr : cnstrs) {  
 Parameter[] params = cnstr.getParameters();  
 ...  
}Copy

If we just need to know the types of the parameters, then Constructor.getParameterTypes() will do the job:

for (Constructor<?> cnstr : cnstrs) {  
 Class<?>[] typesOfParams = cnstr.getParameterTypes();  
 ...  
}Copy

152. Getting the annotation of a receiver type

Starting with JDK 8, we can use explicit *receiver* parameters. Mainly, this means that we can declare an instance method that takes a parameter of the enclosing type with the this Java keyword.

Via explicit *receiver* parameters, we can attach type annotations to this. For example, let's assume that we have the following annotation:

@Target({ElementType.TYPE\_USE})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Ripe {}Copy

Let's use it to annotate this in the eat() method of the Melon class:

public class Melon {  
 ...  
 public void eat(@Ripe Melon this) {}  
 ...  
}Copy

In other words, we can only call the eat() method if the instance of Melon represents a ripe melon:

Melon melon = new Melon("Gac", 2000);  
  
// works only if the melon is ripe  
melon.eat();Copy

Getting the annotation on an explicit *receiver* parameter using reflection can be accomplished via JDK 8 with the java.lang.reflect.Executable.getAnnotatedReceiverType() method. This method is available in the Constructor and Method classes as well, and so we can use it like so:

Class<Melon> clazz = Melon.class;  
Method eatMethod = clazz.getDeclaredMethod("eat");  
  
AnnotatedType annotatedType = eatMethod.getAnnotatedReceiverType();  
  
// modern.challenge.Melon  
System.out.println("Type: " + annotatedType.getType().getTypeName());  
  
// [@modern.challenge.Ripe()]  
System.out.println("Annotations: "   
 + Arrays.toString(annotatedType.getAnnotations()));  
  
// [interface java.lang.reflect.AnnotatedType]  
System.out.println("Class implementing interfaces: "   
 + Arrays.toString(annotatedType.getClass().getInterfaces()));  
  
AnnotatedType annotatedOwnerType   
 = annotatedType.getAnnotatedOwnerType();  
  
// null  
System.out.println("\nAnnotated owner type: " + annotatedOwnerType);Copy

153. Getting synthetic and bridge constructs

By using *synthetic* constructs, we can understand almost any construct that's added by the compiler. More precisely, conforming to the Java language specification: *any constructs introduced by a Java compiler that do not have a corresponding construct in the source code must be marked as synthetic, except for default constructors, the class initialization method, and the values and* *valueOf()* *methods of the* *Enum* *class*.

There are different kinds of *synthetic* constructs (for example, fields, methods, and constructors), but let's take a look at an example of a *synthetic* field. Let's assume that we have the following class:

public class Melon {  
 ...  
 public class Slice {}  
 ...  
}Copy

Notice that we have an inner class called Slice. When the code is compiled, the compiler will alter this class by adding a *synthetic* field that's meant to references the top-level class. This *synthetic* field facilities access to the enclosing class members from a nested class.

In order to check the presence of this *synthetic* field, let's fetch all the declared fields and count them:

Class<Melon.Slice> clazzSlice = Melon.Slice.class;  
Field[] fields = clazzSlice.getDeclaredFields();  
  
// 1  
System.out.println("Number of fields: " + fields.length);Copy

Even if we didn't explicitly declare any fields, notice that one field has been reported. Let's see whether it is *synthetic* and take a look at its name:

// true  
System.out.println("Is synthetic: " + fields[0].isSynthetic());  
  
// this$0  
System.out.println("Name: " + fields[0].getName());Copy

Similar to this example, we can check whether a method or a constructor is *synthetic* via the Method.isSynthetic() and Constructor.isSynthetic() methods.

Now, let's talk about *bridge* methods. These methods are also *synthetic*, and their goal is to handle the *type-erasure* of generics.

Consider the following Melon class:

public class Melon implements Comparator<Melon> {  
  
 @Override  
 public int compare(Melon m1, Melon m2) {  
 return Integer.compare(m1.getWeight(), m2.getWeight());  
 }  
 ...  
}Copy

Here, we implement the Comparator interface and override the compare() method. Moreover, we explicitly specified that the compare() method takes two Melon instances. The compiler will proceed to perform *type-erasure* and create a new method that takes two objects, as follows:

public int compare(Object m1, Object m2) {  
 return compare((Melon) m1, (Melon) m2);  
}Copy

This method is known as a *synthetic bridge* method. We can't see it, but the Java Reflection API can:

Class<Melon> clazz = Melon.class;  
Method[] methods = clazz.getDeclaredMethods();  
Method compareBridge = Arrays.asList(methods).stream()  
 .filter(m -> m.isSynthetic() && m.isBridge())  
 .findFirst()  
 .orElseThrow();  
  
// public int modern.challenge.Melon.compare(  
// java.lang.Object, java.lang.Object)  
System.out.println(compareBridge);Copy

154. Checking the variable number of arguments

In Java, a method can receive a variable number of arguments if its signature contains an argument of the varargs type.

For example, the plantation() method takes a variable number of arguments, for example, Seed... seeds:

public class Melon {  
 ...  
 public void plantation(String type, Seed...seeds) {}  
 ...  
}Copy

Now, the Java Reflection API can tell whether this method supports a variable number of arguments via the Method.isVarArgs() method, as follows:

Class<Melon> clazz = Melon.class;  
Method[] methods = clazz.getDeclaredMethods();  
  
for (Method method: methods) {  
 System.out.println("Method name: " + method.getName()   
 + " varargs? " + method.isVarArgs());  
}Copy

You will receive output similar to the following:

Method name: plantation, varargs? true  
Method name: getWeight, varargs? false  
Method name: toString, varargs? false  
Method name: getType, varargs? falseCopy

155. Checking default methods

Java 8 has enriched the concept of interfaces with default methods. These methods are written inside interfaces and have a default implementation. For example, the Slicer interface has a default method called slice():

public interface Slicer {  
  
 public void type();  
  
 default void slice() {  
 System.out.println("slice");  
 }  
}Copy

Now, any implementation of Slicer must implement the type() method and, optionally, can override the slice() method or rely on the default implementation.

The Java Reflection API can identify a default method via the Method.isDefault() flag method:

Class<Slicer> clazz = Slicer.class;  
Method[] methods = clazz.getDeclaredMethods();  
  
for (Method method: methods) {  
 System.out.println("Method name: " + method.getName()   
 + ", is default? " + method.isDefault());  
}Copy

We will receive the following output:

Method name: type, is default? false  
Method name: slice, is default? trueCopy

156. Nest-based access control via reflection

Among the features of JDK 11, we have several *hotspots* (changes at bytecode level). One of these *hotspots* is known as JEP 181, or **nest-based access control** (**nests**). Basically, the *nest* term defines a new access control context that *allows classes that are logically part of the same code entity, but which are compiled with distinct class files, to access each other's private members without the need for compilers to insert accessibility-broadening bridge methods.*

So, in other words, *nests* allow nested classes to be compiled to different class files that belong to the same enclosing class. These are then allowed to access each other's private classes without the use of *synthetic*/*bridge* methods.

Let's consider the following code:

public class Car {  
  
 private String type = "Dacia";  
  
 public class Engine {  
  
 private String power = "80 hp";  
  
 public void addEngine() {  
 System.out.println("Add engine of " + power   
 + " to car of type " + type);  
 }  
 }  
}Copy

Let's run javap (the Java class file disassembler tool that allows us to analyze the bytecode) for Car.class in JDK 10. The following screenshot highlights the important part of this code:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

As we can see, to access the enclosing class field, Car.type, from the Engine.addEngine() method, Java has altered the code and added a *bridge*package-private method known as access$000(). Mainly, this is synthetically generated and can be seen via reflection using the Method.isSynthetic() and Method.isBridge() methods.

Even if we see (or perceive) the Car (outer) and Engine (nested) classes as being in the same class, they are compiled to different files (Car.class and Car$Engine.class). Conforming to this statement, our expectations imply that the outer and the nested classes can access each other's private members.

But being in separate files, this is not possible. In order to sustain our expectations, Java adds the *synthetic bridge*package-private method, access$000().

However, Java 11 introduces the *nests* access control context, which provides support for private access within outer and nested classes. This time, the outer and nested classes are linked to two attributes and they form a *nest* (we say that they are *nestmates*). Mainly, nested classes are linked to the NestMembers attribute, while the outer class is linked to the NestHost attribute. No extra *synthetic* method is generated.

In the following screenshot, we can see javap being executed in JDK 11 for Car.class (notice the NestMembers attribute):

Texto

Descripción generada automáticamente

The following screenshot shows the javap output in JDK 11 for Car$Engine.class (notice the NestHost attribute):

Texto

Descripción generada automáticamente

Access via the Reflection API

Without nest-based access control, reflection capabilities are also limited. For example, before JDK 11, the following snippet of code would throw IllegalAccessException:

Car newCar = new Car();  
Engine engine = newCar.new Engine();  
  
Field powerField = Engine.class.getDeclaredField("power");  
powerField.set(engine, power);Copy

We can allow access by explicitly calling powerField.setAccessible(true):

...  
Field powerField = Engine.class.getDeclaredField("power");  
powerField.setAccessible(true);  
powerField.set(engine, power);  
...Copy

Starting with JDK 11, there is no need to call setAccessible().

Moreover, JDK 11 comes with three methods that enrich the Java Reflection API with support for *nests*. These methods are Class.getNestHost(), Class.getNestMembers(), and Class.isNestmateOf().

Let's consider the following Melon class with several nested classes (Slice, Peeler, and Juicer):

public class Melon {  
 ...  
 public class Slice {  
 public class Peeler {}  
 }  
  
 public class Juicer {}  
 ...  
}Copy

Now, let's define a Class for each of them:

Class<Melon> clazzMelon = Melon.class;  
Class<Melon.Slice> clazzSlice = Melon.Slice.class;  
Class<Melon.Juicer> clazzJuicer = Melon.Juicer.class;  
Class<Melon.Slice.Peeler> clazzPeeler = Melon.Slice.Peeler.class;Copy

In order to see NestHost of each class, we need to call Class.getNestHost():

// class modern.challenge.Melon  
Class<?> nestClazzOfMelon = clazzMelon.getNestHost();  
  
// class modern.challenge.Melon  
Class<?> nestClazzOfSlice = clazzSlice.getNestHost();  
  
// class modern.challenge.Melon  
Class<?> nestClazzOfPeeler = clazzPeeler.getNestHost();  
  
// class modern.challenge.Melon  
Class<?> nestClazzOfJuicer = clazzJuicer.getNestHost();Copy

Two things should be highlighted here. First, note that NestHost of Melon is Melon itself. Second, note that NestHost of Peeler is Melon, not Slice. Since Peeler is an inner class of Slice, we may think that its NestHost is Slice, but this assumption is not true.

Now, let's list NestMembers of each class:

Class<?>[] nestMembersOfMelon = clazzMelon.getNestMembers();  
Class<?>[] nestMembersOfSlice = clazzSlice.getNestMembers();  
Class<?>[] nestMembersOfJuicer = clazzJuicer.getNestMembers();  
Class<?>[] nestMembersOfPeeler = clazzPeeler.getNestMembers();Copy

All of them will return same NestMembers:

[class modern.challenge.Melon, class modern.challenge.Melon$Juicer, class modern.challenge.Melon$Slice, class modern.challenge.Melon$Slice$Peeler]Copy

Finally, let's check *nestmates*:

boolean melonIsNestmateOfSlice   
 = clazzMelon.isNestmateOf(clazzSlice); // true  
  
boolean melonIsNestmateOfJuicer   
 = clazzMelon.isNestmateOf(clazzJuicer); // true  
  
boolean melonIsNestmateOfPeeler   
 = clazzMelon.isNestmateOf(clazzPeeler); // true  
  
boolean sliceIsNestmateOfJuicer   
 = clazzSlice.isNestmateOf(clazzJuicer); // true  
  
boolean sliceIsNestmateOfPeeler   
 = clazzSlice.isNestmateOf(clazzPeeler); // true  
  
boolean juicerIsNestmateOfPeeler   
 = clazzJuicer.isNestmateOf(clazzPeeler); // trueCopy

157. Reflection for getters and setters

Just as a quick reminder, getters and setters are methods (also known as accessors) that are used for accessing the fields of a class (for example, private fields).

First, let's see how we can fetch the existing getters and setters. Later on, we will try to generate the missing getters and setters via reflection.

Fetching getters and setters

Mainly, there are several solutions for obtaining the getters and setters of a class via reflection. Let's assume that we want to fetch the getters and setters of the following Melon class:

public class Melon {  
  
 private String type;  
 private int weight;  
 private boolean ripe;  
 ...  
  
 public String getType() {  
 return type;  
 }  
  
 public void setType(String type) {  
 this.type = type;  
 }  
  
 public int getWeight() {  
 return weight;  
 }  
  
 public void setWeight(int weight) {  
 this.weight = weight;  
 }  
  
 public boolean isRipe() {  
 return ripe;  
 }  
  
 public void setRipe(boolean ripe) {  
 this.ripe = ripe;  
 }  
 ...  
}Copy

Let's start with a solution that gets all the declared methods of a class via reflection (for example, via Class.getDeclaredMethods()). Now, loop Method[] and filter it by constraints that are specific to getters and setters (for example, start with the get/set prefix, return void or a certain type, and so on).

Another solution is getting all the declared fields of a class via reflection (for example, via Class.getDeclaredFields()). Now, loop Field[] and try to obtain the getters and setters via Class.getDeclaredMethod() by passing the name of the field (prefixed with get/set/is and the first letter capitalized) and the type of the field (in the case of setters) to it.

Finally, a more elegant solution will rely on the PropertyDescriptor and Introspector APIs. These APIs are available in the java.beans.\* package and are dedicated to working with JavaBeans.

Many of the features that are exposed by these two classes rely on reflection behind the scene.

The PropertyDescriptor class can return the method that's used for reading a JavaBean property via getReadMethod(). Moreover, it can return the method that's used for writing a JavaBean property via getWriteMethod(). Relying on these two methods, we can fetch the getters and setters of the Melon class, as follows:

for (PropertyDescriptor pd:  
 Introspector.getBeanInfo(Melon.class).getPropertyDescriptors()) {  
  
 if (pd.getReadMethod() != null && !"class".equals(pd.getName())) {  
 System.out.println(pd.getReadMethod());  
 }  
  
 if (pd.getWriteMethod() != null && !"class".equals(pd.getName())) {  
 System.out.println(pd.getWriteMethod());  
 }  
}Copy

The output is as follows:

public boolean modern.challenge.Melon.isRipe()  
public void modern.challenge.Melon.setRipe(boolean)  
public java.lang.String modern.challenge.Melon.getType()  
public void modern.challenge.Melon.setType(java.lang.String)  
public int modern.challenge.Melon.getWeight()  
public void modern.challenge.Melon.setWeight(int)Copy

Now, let's assume that we have the following Melon instance:

Melon melon = new Melon("Gac", 1000);Copy

Here, we want to call the getType() getter:

// the returned type is Gac  
Object type = new PropertyDescriptor("type",  
 Melon.class).getReadMethod().invoke(melon);Copy

Now, let's call the setWeight() setter:

// set weight of Gac to 2000  
new PropertyDescriptor("weight", Melon.class)  
 .getWriteMethod().invoke(melon, 2000);Copy

Calling for an inexistent property will cause IntrospectionException:

try {  
 Object shape = new PropertyDescriptor("shape",  
 Melon.class).getReadMethod().invoke(melon);  
 System.out.println("Melon shape: " + shape);  
} catch (IntrospectionException e) {  
 System.out.println("Property not found: " + e);  
}Copy

Generating getters and setters

Let's assume that Melon has three fields (type, weight, and ripe) and defines only a getter for type and a setter for ripe:

public class Melon {  
  
 private String type;  
 private int weight;  
 private boolean ripe;  
 ...  
  
 public String getType() {  
 return type;  
 }  
  
 public void setRipe(boolean ripe) {  
 this.ripe = ripe;  
 }  
 ...  
}Copy

In order to generate the missing getters and setters, we start by identifying them. The following solution loops the declared fields of the given class and assumes that a foo field doesn't have a getter if the following apply:

* There is no get/isFoo() method
* The return type is not the same as the field type
* The number of arguments is not 0

For each missing getter, this solution adds in a map an entry containing the field name and type:

private static Map<String, Class<?>>   
 fetchMissingGetters(Class<?> clazz) {  
  
 Map<String, Class<?>> getters = new HashMap<>();  
 Field[] fields = clazz.getDeclaredFields();  
 String[] names = new String[fields.length];  
 Class<?>[] types = new Class<?>[fields.length];  
  
 Arrays.setAll(names, i -> fields[i].getName());  
 Arrays.setAll(types, i -> fields[i].getType());  
  
 for (int i = 0; i < names.length; i++) {  
 String getterAccessor = fetchIsOrGet(names[i], types[i]);  
  
 try {  
 Method getter = clazz.getDeclaredMethod(getterAccessor);  
 Class<?> returnType = getter.getReturnType();  
  
 if (!returnType.equals(types[i]) ||  
 getter.getParameterCount() != 0) {  
 getters.put(names[i], types[i]);  
 }  
 } catch (NoSuchMethodException ex) {  
 getters.put(names[i], types[i]);  
 // log exception  
 }  
 }  
  
 return getters;  
}Copy

Further, the solution loops the declared fields of the given class and assume that a *foo* field doesn't have a setter if the following apply:

* The field is not final
* There is no setFoo() method
* The method returns void
* The method has a single parameter
* The parameter type is the same as the field type
* If the parameter name is present, it should be the same as the name of the field

For each missing setter, this solution adds an entry containing the field name and type in a map:

private static Map<String, Class<?>>   
 fetchMissingSetters(Class<?> clazz) {  
  
 Map<String, Class<?>> setters = new HashMap<>();  
 Field[] fields = clazz.getDeclaredFields();  
 String[] names = new String[fields.length];  
 Class<?>[] types = new Class<?>[fields.length];  
  
 Arrays.setAll(names, i -> fields[i].getName());  
 Arrays.setAll(types, i -> fields[i].getType());  
  
 for (int i = 0; i < names.length; i++) {  
 Field field = fields[i];  
 boolean finalField = !Modifier.isFinal(field.getModifiers());  
  
 if (finalField) {  
 String setterAccessor = fetchSet(names[i]);  
  
 try {  
 Method setter = clazz.getDeclaredMethod(  
 setterAccessor, types[i]);  
  
 if (setter.getParameterCount() != 1 ||  
 !setter.getReturnType().equals(void.class)) {  
  
 setters.put(names[i], types[i]);  
 continue;  
 }  
  
 Parameter parameter = setter.getParameters()[0];  
 if ((parameter.isNamePresent() &&  
 !parameter.getName().equals(names[i])) ||  
 !parameter.getType().equals(types[i])) {  
 setters.put(names[i], types[i]);  
 }  
 } catch (NoSuchMethodException ex) {  
 setters.put(names[i], types[i]);  
 // log exception  
 }  
 }  
 }  
  
 return setters;  
}Copy

Well, so far, we know what fields don't have getters and setters. Their names and types are stored in a map. Let's loop the map and generate the getters:

public static StringBuilder generateGetters(Class<?> clazz) {  
  
 StringBuilder getterBuilder = new StringBuilder();  
 Map<String, Class<?>> accessors = fetchMissingGetters(clazz);  
  
 for (Entry<String, Class<?>> accessor: accessors.entrySet()) {  
 Class<?> type = accessor.getValue();  
 String field = accessor.getKey();  
 String getter = fetchIsOrGet(field, type);  
  
 getterBuilder.append("\npublic ")  
 .append(type.getSimpleName()).append(" ")  
 .append(getter)  
 .append("() {\n")  
 .append("\treturn ")  
 .append(field)  
 .append(";\n")  
 .append("}\n");  
 }  
  
 return getterBuilder;  
}Copy

And let's generate the setters:

public static StringBuilder generateSetters(Class<?> clazz) {  
  
 StringBuilder setterBuilder = new StringBuilder();  
 Map<String, Class<?>> accessors = fetchMissingSetters(clazz);  
  
 for (Entry<String, Class<?>> accessor: accessors.entrySet()) {  
 Class<?> type = accessor.getValue();  
 String field = accessor.getKey();  
 String setter = fetchSet(field);  
  
 setterBuilder.append("\npublic void ")  
 .append(setter)  
 .append("(").append(type.getSimpleName()).append(" ")  
 .append(field).append(") {\n")  
 .append("\tthis.")  
 .append(field).append(" = ")  
 .append(field)  
 .append(";\n")  
 .append("}\n");  
 }  
  
 return setterBuilder;  
}Copy

The preceding solution relies on three simple helpers listed in the following. The code is straightforward:

private static String fetchIsOrGet(String name, Class<?> type) {  
 return "boolean".equalsIgnoreCase(type.getSimpleName()) ?  
 "is" + uppercase(name) : "get" + uppercase(name);  
}  
  
private static String fetchSet(String name) {  
 return "set" + uppercase(name);  
}  
  
private static String uppercase(String name) {  
 return name.substring(0, 1).toUpperCase() + name.substring(1);  
}Copy

Now, let's call it for the Melon class:

Class<?> clazz = Melon.class;  
StringBuilder getters = generateGetters(clazz);  
StringBuilder setters = generateSetters(clazz);Copy

The output will reveal the following generated getters and setters:

public int getWeight() {  
 return weight;  
}  
  
public boolean isRipe() {  
 return ripe;  
}  
  
public void setWeight(int weight) {  
 this.weight = weight;  
}  
  
public void setType(String type) {  
 this.type = type;  
}Copy

158. Reflecting annotations

Java annotations have got a lot of attention from the Java Reflection API. Let's see several solutions for inspecting several kinds of annotations (for example, package, class, and method).

Mainly, all major Reflection API classes that represent artifacts that support annotation (for example, Package, Constructor, Class, Method, and Field) reveal a set of common methods for working with annotations. Common methods include:

* getAnnotations(): Return all annotations specific to a certain artifact
* getDeclaredAnnotations(): Return all annotations declared directly to a certain artifact
* getAnnotation(): Return an annotation by type
* getDeclaredAnnotation(): Return an annotation by type declared directly to a certain artifact (JDK 1.8)
* getDeclaredAnnotationsByType(): Return all annotations by type declared directly to a certain artifact (JDK 1.8)
* isAnnotationPresent(): Return true if an annotation for the specified type is found on the given artifact

getAnnotatedReceiverType() was discussed earlier in the *Get annotation on receiver type* section.

In the next sections, let's talk about inspecting annotations of packages, classes, methods, and so on.

Inspecting package annotations

Annotations specific to packages are added in package-info.java as in the following screenshot. Here, the modern.challenge package was annotated with the @Packt annotation:

Forma

Descripción generada automáticamente con confianza media

A convenient solution to inspect the annotations of a package starts from one of its classes. For example, if, in this package (modern.challenge), we have the Melon class, then we can obtain all annotations of this package as follows:

Class<Melon> clazz = Melon.class;  
Annotation[] pckgAnnotations = clazz.getPackage().getAnnotations();Copy

Annotation[] printed via Arrays.toString() reveals a single result:

[@modern.challenge.Packt()]Copy

Inspecting class annotations

The Melon class has a single annotation, @Fruit:

Imagen que contiene Texto

Descripción generada automáticamente

But we can fetch all of them via getAnnotations():

Class<Melon> clazz = Melon.class;  
Annotation[] clazzAnnotations = clazz.getAnnotations();Copy

The returned array printed via Arrays.toString() reveals a single result:

[@modern.challenge.Fruit(name="melon", value="delicious")]Copy

In order to access the name and value attributes of an annotation, we can cast it as follows:

Fruit fruitAnnotation = (Fruit) clazzAnnotations[0];  
System.out.println("@Fruit name: " + fruitAnnotation.name());  
System.out.println("@Fruit value: " + fruitAnnotation.value());Copy

Or we can use the getDeclaredAnnotation() method to fetch the right type directly:

Fruit fruitAnnotation = clazz.getDeclaredAnnotation(Fruit.class);Copy

Inspecting methods annotations

Let's inspect the @Ripe annotation of the eat() method from the Melon class:

Imagen que contiene Texto

Descripción generada automáticamente

First, let's fetch all the declared annotations, and afterward, let's resume to @Ripe:

Class<Melon> clazz = Melon.class;  
Method methodEat = clazz.getDeclaredMethod("eat");  
Annotation[] methodAnnotations = methodEat.getDeclaredAnnotations();Copy

The returned array printed via Arrays.toString() reveals a single result:

[@modern.challenge.Ripe(value=true)]Copy

And let's cast methodAnnotations[0] to Ripe:

Ripe ripeAnnotation = (Ripe) methodAnnotations[0];  
System.out.println("@Ripe value: " + ripeAnnotation.value());Copy

Or we can use the getDeclaredAnnotation() method to fetch the right type directly:

Ripe ripeAnnotation = methodEat.getDeclaredAnnotation(Ripe.class);Copy

Inspecting annotations of the thrown exceptions

For inspecting the annotations of the thrown exceptions, we need to call the getAnnotatedExceptionTypes() method:

Forma

Descripción generada automáticamente con confianza baja

This method returns the thrown exceptions types included those that are annotated:

Class<Melon> clazz = Melon.class;  
Method methodEat = clazz.getDeclaredMethod("eat");  
AnnotatedType[] exceptionsTypes   
 = methodEat.getAnnotatedExceptionTypes();Copy

The returned array printed via Arrays.toString() reveals a single result:

[@modern.challenge.Runtime() java.lang.IllegalStateException]Copy

Extracting the first exception type can be accomplished as follows:

// class java.lang.IllegalStateException  
System.out.println("First exception type: "  
 + exceptionsTypes[0].getType());Copy

Extracting the annotations of the first exception type can be done as follows:

// [@modern.challenge.Runtime()]  
System.out.println("Annotations of the first exception type: "   
 + Arrays.toString(exceptionsTypes[0].getAnnotations()));Copy

Inspecting annotations of the return type

For inspecting the annotations of a method return, we need to call the getAnnotatedReturnType() method:

Imagen que contiene Diagrama

Descripción generada automáticamente

This method returns the annotated return type of the given method:

Class<Melon> clazz = Melon.class;  
Method methodSeeds = clazz.getDeclaredMethod("seeds");  
AnnotatedType returnType = methodSeeds.getAnnotatedReturnType();  
  
// java.util.List<modern.challenge.Seed>  
System.out.println("Return type: "   
 + returnType.getType().getTypeName());  
  
// [@modern.challenge.Shape(value="oval")]  
System.out.println("Annotations of the return type: "   
 + Arrays.toString(returnType.getAnnotations()));Copy

Inspecting annotations of the method's parameters

Having a method, we can inspect the annotations of its parameters by calling getParameterAnnotations():

Imagen que contiene Texto

Descripción generada automáticamente

This method returns a matrix (array of arrays) containing the annotations on the formal parameters, in declaration order:

Class<Melon> clazz = Melon.class;  
Method methodSlice = clazz.getDeclaredMethod("slice", int.class);  
Annotation[][] paramAnnotations   
 = methodSlice.getParameterAnnotations();Copy

Fetching each parameter type with its annotations (in this case, we have an int parameter with two annotations) can be accomplished via getParameterTypes(). Since this method maintains the declaration order as well, we can extract some information, as follows:

Class<?>[] parameterTypes = methodSlice.getParameterTypes();  
  
int i = 0;  
for (Annotation[] annotations: paramAnnotations) {  
 Class parameterType = parameterTypes[i++];  
 System.out.println("Parameter: " + parameterType.getName());  
  
 for (Annotation annotation: annotations) {  
 System.out.println("Annotation: " + annotation);  
 System.out.println("Annotation name: "   
 + annotation.annotationType().getSimpleName());  
 }  
}Copy

And, the output should be as follows:

Parameter type: int  
Annotation: @modern.challenge.Ripe(value=true)  
Annotation name: Ripe  
Annotation: @modern.challenge.Shape(value="square")  
Annotation name: ShapeCopy

Inspecting annotations of fields

Having a field, we can fetch its annotations via getDeclaredAnnotations():

Imagen que contiene Logotipo

Descripción generada automáticamente

Here it is the code:

Class<Melon> clazz = Melon.class;  
Field weightField = clazz.getDeclaredField("weight");  
Annotation[] fieldAnnotations = weightField.getDeclaredAnnotations();Copy

Getting the value of the @Unit annotation can be done as follows:

Unit unitFieldAnnotation = (Unit) fieldAnnotations[0];  
System.out.println("@Unit value: " + unitFieldAnnotation.value());Copy

Or, use the getDeclaredAnnotation() method to fetch the right type directly:

Unit unitFieldAnnotation   
 = weightField.getDeclaredAnnotation(Unit.class);Copy

Inspecting annotations of the superclass

For inspecting the annotations of the superclass, we need to call the getAnnotatedSuperclass() method:

Texto

Descripción generada automáticamente con confianza baja

This method returns the superclass type that is annotated:

Class<Melon> clazz = Melon.class;  
AnnotatedType superclassType = clazz.getAnnotatedSuperclass();Copy

And let's get some information as well:

// modern.challenge.Cucurbitaceae  
 System.out.println("Superclass type: "   
 + superclassType.getType().getTypeName());  
  
 // [@modern.challenge.Family()]  
 System.out.println("Annotations: "   
 + Arrays.toString(superclassType.getDeclaredAnnotations()));  
  
 System.out.println("@Family annotation present: "   
 + superclassType.isAnnotationPresent(Family.class)); // trueCopy

Inspecting annotations of interfaces

For inspecting the annotations of the implemented interfaces, we need to call the getAnnotatedInterfaces() method:

Texto

Descripción generada automáticamente

This method returns the interfaces types that are annotated:

Class<Melon> clazz = Melon.class;  
AnnotatedType[] interfacesTypes = clazz.getAnnotatedInterfaces();Copy

The returned array printed via Arrays.toString() reveals a single result:

[@modern.challenge.ByWeight() java.lang.Comparable]Copy

Extracting the first interface type can be accomplished as follows:

// interface java.lang.Comparable  
System.out.println("First interface type: "   
 + interfacesTypes[0].getType());Copy

Moreover, extracting the annotations of the first interface type can be done as follows:

// [@modern.challenge.ByWeight()]  
System.out.println("Annotations of the first exception type: "   
 + Arrays.toString(interfacesTypes[0].getAnnotations()));Copy

Get annotations by type

Having multiple annotations of the same type on certain components, we can fetch all of them via getAnnotationsByType(). For a class, we can do it as follows:

Class<Melon> clazz = Melon.class;  
Fruit[] clazzFruitAnnotations   
 = clazz.getAnnotationsByType(Fruit.class);Copy

Get a declared annotation

Trying to fetch by type a single annotation declared directly on a certain artifact can be done as shown in the following example:

Class<Melon> clazz = Melon.class;  
Method methodEat = clazz.getDeclaredMethod("eat");  
Ripe methodRipeAnnotation   
 = methodEat.getDeclaredAnnotation(Ripe.class);Copy

159. Invoking an instance method

Let's assume that we have the following Melon class:

public class Melon {  
 ...  
 public Melon() {}  
  
 public List<Melon> cultivate(  
 String type, Seed seed, int noOfSeeds) {  
  
 System.out.println("The cultivate() method was invoked ...");  
  
 return Collections.nCopies(noOfSeeds, new Melon("Gac", 5));  
 }  
 ...  
}Copy

Our goal is to invoke the cultivate() method and obtain the return via the Java Reflection API.

First, let's fetch the cultivate() method as a Method via Method.getDeclaredMethod(). All we have to do is pass the name of the method (in this case, cultivate()) and the right types of parameters (String, Seed, and int) to getDeclaredMethod(). Second argument of getDeclaredMethod() is a varargs of Class<?> type, therefore, it can be empty for methods with no parameters or contain the list of parameters types as in the following example:

Method cultivateMethod = Melon.class.getDeclaredMethod(  
 "cultivate", String.class, Seed.class, int.class);Copy

Then, let's obtain an instance of the Melon class. We want to invoke an instance method; therefore, we need an instance. Relying on the empty constructor of Melon and the Java Reflection API, we can do it as follows:

Melon instanceMelon = Melon.class  
 .getDeclaredConstructor().newInstance();Copy

Finally, we focus on the Method.invoke() method. Mainly, we need to pass to this method the instance used for calling the cultivate() method and some values for the parameters:

List<Melon> cultivatedMelons = (List<Melon>) cultivateMethod.invoke(  
 instanceMelon, "Gac", new Seed(), 10);Copy

The success of invocation is revealed by the following message:

The cultivate() method was invoked ...Copy

Moreover, if we print the return of invocation via System.out.println(), then we get the following result:

[Gac(5g), Gac(5g), Gac(5g), ...]Copy

We've just cultivated 10 gacs via reflection.

160. Getting static methods

Let's assume that we have the following Melon class:

public class Melon {  
 ...  
 public void eat() {}  
  
 public void weighsIn() {}  
  
 public static void cultivate(Seed seeds) {  
 System.out.println("The cultivate() method was invoked ...");  
 }  
  
 public static void peel(Slice slice) {  
 System.out.println("The peel() method was invoked ...");  
 }  
  
 // getters, setters, toString() omitted for brevity  
}Copy

This class has two static methods—cultivate() and peel(). Let's fetch these two methods in List<Method>.

The solution to this problem has two main steps:

1. Fetch all the available methods of the given class
2. Filter those that contain the static modifier via the Modifier.isStatic() method

In code, it looks like this:

List<Method> staticMethods = new ArrayList<>();  
  
Class<Melon> clazz = Melon.class;  
Method[] methods = clazz.getDeclaredMethods();  
  
for (Method method: methods) {  
  
 if (Modifier.isStatic(method.getModifiers())) {  
 staticMethods.add(method);  
 }  
}Copy

The result of printing the list via System.out.println() is as follows:

[public static void   
 modern.challenge.Melon.peel(modern.challenge.Slice),  
  
 public static void   
 modern.challenge.Melon.cultivate(modern.challenge.Seed)]Copy

One step further, and we may want to call one of these two methods.

For example, let's call the peel() method (notice that we pass null instead of an instance of Melon since a static method doesn't need an instance):

Method method = clazz.getMethod("peel", Slice.class);  
method.invoke(null, new Slice());Copy

The output signals that the peel() method was successfully invoked:

The peel() method was invoked ...Copy

161. Getting generic types of method, fields, and exceptions

Let's assume that we have the following Melon class (listed are only the parts relevant to this problem):

public class Melon<E extends Exception>  
 extends Fruit<String, Seed> implements Comparable<Integer> {  
  
 ...  
 private List<Slice> slices;  
 ...  
  
 public List<Slice> slice() throws E {  
 ...  
 }  
  
 public Map<String, Integer> asMap(List<Melon> melons) {  
 ...  
 }  
 ...  
}Copy

The Melon class contains several generic types associated with different artifacts. Mainly, the generic types of super classes, interfaces, classes, methods, and fields are ParameterizedType instances. For each ParameterizedType, we need to fetch the actual type of arguments via ParameterizedType.getActualTypeArguments(). The Type[] returned by this method can be iterated to extract information about each argument, as follows:

public static void printGenerics(Type genericType) {  
  
 if (genericType instanceof ParameterizedType) {  
 ParameterizedType type = (ParameterizedType) genericType;  
 Type[] typeOfArguments = type.getActualTypeArguments();  
  
 for (Type typeOfArgument: typeOfArguments) {  
 Class classTypeOfArgument = (Class) typeOfArgument;  
 System.out.println("Class of type argument: "   
 + classTypeOfArgument);  
  
 System.out.println("Simple name of type argument: "   
 + classTypeOfArgument.getSimpleName());  
 }  
 }  
}Copy

Now, let's see how we can deal with generics of methods.

Generics of methods

For example, let's get the generic return types for the slice() and asMap() methods. This can be accomplished via the Method.getGenericReturnType() method as follows:

Class<Melon> clazz = Melon.class;  
  
Method sliceMethod = clazz.getDeclaredMethod("slice");  
Method asMapMethod = clazz.getDeclaredMethod("asMap", List.class);  
  
Type sliceReturnType = sliceMethod.getGenericReturnType();  
Type asMapReturnType = asMapMethod.getGenericReturnType();Copy

Now, calling printGenerics(sliceReturnType) will output the following:

Class of type argument: class modern.challenge.Slice  
Simple name of type argument: SliceCopy

And, calling printGenerics(asMapReturnType) will output the following:

Class of type argument: class java.lang.String  
Simple name of type argument: String  
  
Class of type argument: class java.lang.Integer  
Simple name of type argument: IntegerCopy

Generic parameters of methods can be obtained via Method.getGenericParameterTypes(), as follows:

Type[] asMapParamTypes = asMapMethod.getGenericParameterTypes();Copy

Further, we call printGenerics() for each Type (each generic parameter):

for (Type paramType: asMapParamTypes) {  
 printGenerics(paramType);  
}Copy

Following is the output (there is a single generic parameter, List<Melon>):

Class of type argument: class modern.challenge.Melon  
Simple name of type argument: MelonCopy

Generics of fields

In the case of fields (for example, slices), generics can be fetched via Field.getGenericType(), as follows:

Field slicesField = clazz.getDeclaredField("slices");  
Type slicesType = slicesField.getGenericType();Copy

Calling printGenerics(slicesType) will output the following:

Class of type argument: class modern.challenge.Slice  
Simple name of type argument: SliceCopy

Generics of a superclass

Getting the generics of a superclass can be accomplished by calling the getGenericSuperclass() method of the current class:

Type superclassType = clazz.getGenericSuperclass();Copy

Calling printGenerics(superclassType) will output the following:

Class of type argument: class java.lang.String  
Simple name of type argument: String  
  
Class of type argument: class modern.challenge.Seed  
Simple name of type argument: SeedCopy

Generics of interfaces

Getting the generics of implemented interfaces can be accomplished by calling the getGenericInterfaces() method of the current class:

Type[] interfacesTypes = clazz.getGenericInterfaces();Copy

Further, we call printGenerics() for each Type. Following is the output (there is a single interface, Comparable<Integer>):

Class of type argument: class java.lang.Integer  
Simple name of type argument: IntegerCopy

Generics of exceptions

Generic types of exceptions are materialized in instances of TypeVariable or ParameterizedType. This time, the helper method for extracting and printing information about generics based on TypeVariable can be written as follows:

public static void printGenericsOfExceptions(Type genericType) {  
  
 if (genericType instanceof TypeVariable) {  
 TypeVariable typeVariable = (TypeVariable) genericType;  
 GenericDeclaration genericDeclaration  
 = typeVariable.getGenericDeclaration();  
  
 System.out.println("Generic declaration: " + genericDeclaration);  
  
 System.out.println("Bounds: ");  
 for (Type type: typeVariable.getBounds()) {  
 System.out.println(type);  
 }  
 }  
}Copy

Having this helper, we can pass to it the exceptions thrown by a method via getGenericExceptionTypes(). If an exception type is a type variable (TypeVariable) or a parameterized type (ParameterizedType), it is created. Otherwise, it is resolved:

Type[] exceptionsTypes = sliceMethod.getGenericExceptionTypes();Copy

Further, we call the printGenerics() for each Type:

for (Type paramType: exceptionsTypes) {  
 printGenericsOfExceptions(paramType);  
}Copy

The output will be as follows:

Generic declaration: class modern.challenge.Melon  
Bounds: class java.lang.ExceptionCopy

Most probably, printing the extracted information about generics will not be useful, therefore, feel free to adapt the preceding helpers based on your needs. For example, collect the information and return it as List, Map, and so on.

162. Getting public and private fields

The solution to this problem relies on the Modifier.isPublic() and Modifier.isPrivate() methods.

Let's assume the following Melon class has two public fields and two private fields:

public class Melon {  
  
 private String type;  
 private int weight;  
  
 public Peeler peeler;  
 public Juicer juicer;  
 ...  
}Copy

First, we need to fetch the Field[] array corresponding to this class via the getDeclaredFields() method:

Class<Melon> clazz = Melon.class;  
Field[] fields = clazz.getDeclaredFields();Copy

Field[] contains all the four fields from earlier. Further, let's iterate this array and let's apply Modifier.isPublic() and Modifier.isPrivate() flag methods to each Field:

List<Field> publicFields = new ArrayList<>();  
List<Field> privateFields = new ArrayList<>();  
  
for (Field field: fields) {  
 if (Modifier.isPublic(field.getModifiers())) {  
 publicFields.add(field);  
 }  
  
 if (Modifier.isPrivate(field.getModifiers())) {  
 privateFields.add(field);  
 }  
}Copy

The publicFields list contains only public fields, and the privateFields list contains only private fields. If we quickly print these two lists via System.out.println(), then the output will be as follows:

Public fields:  
[public modern.challenge.Peeler modern.challenge.Melon.peeler,  
public modern.challenge.Juicer modern.challenge.Melon.juicer]  
  
Private fields:  
[private java.lang.String modern.challenge.Melon.type,  
private int modern.challenge.Melon.weight]Copy

163. Working with arrays

The Java Reflection API comes with a class dedicated to working with arrays. This class is named java.lang.reflect.Array.

For example, the following snippet of code creates an array of int. The first parameter tells what type each element in the array should be of. The second parameter represents the length of the array. Therefore, an array of 10 integers can be defined via Array.newInstance() as follows:

int[] arrayOfInt = (int[]) Array.newInstance(int.class, 10);Copy

Using Java Reflection, we can alter the content of an array. There is a general set() method and a bunch of set*Foo*() methods (for example, setInt(), and setFloat()). Setting the value at index 0 to 100 can be done as follows:

Array.setInt(arrayOfInt, 0, 100);Copy

Fetching a value from an array can be accomplished via the get() and getFoo() methods (these methods get the array and the index as arguments and return the value from the specified index):

int valueIndex0 = Array.getInt(arrayOfInt, 0);Copy

Getting the Class of an array can be done as follows:

Class<?> stringClass = String[].class;  
Class<?> clazz = arrayOfInt.getClass();Copy

And we can extract the type of the array via getComponentType():

// int  
Class<?> typeInt = clazz.getComponentType();  
  
// java.lang.String  
Class<?> typeString = stringClass.getComponentType();Copy

164. Inspecting modules

Java 9 has added the concept of *modules* via the Java Platform Module System. Basically, a module is a set of packages managed by that module (for example, the module decides which packages are visible outside the module).

An application with two modules can be shaped as in the following screenshot:

Diagrama

Descripción generada automáticamente con confianza baja

There are two modules—org.player and org.tournament. The org.player module requires the org.tournament module, and the org.tournament module exports the com.management package.

Java Reflection API represents a module via the java.lang.Module class (in the java.base module). Via the Java Reflection API, we can extract information or modify a module.

For start, we can obtain a Module instance as in the following two examples:

Module playerModule = Player.class.getModule();  
Module managerModule = Manager.class.getModule();Copy

The name of a module can be obtained via the Module.getName() method:

// org.player  
System.out.println("Class 'Player' is in module: "   
 + playerModule.getName());  
  
// org.tournament  
System.out.println("Class 'Manager' is in module: "   
 + managerModule.getName());Copy

Having a Module instance, we can call several methods for getting different information. For example, we can find out whether a module is named or it has exported or opened a certain package:

boolean playerModuleIsNamed = playerModule.isNamed(); // true  
boolean managerModuleIsNamed = managerModule.isNamed(); // true  
  
boolean playerModulePnExported   
 = playerModule.isExported("com.members"); // false  
boolean managerModulePnExported   
 = managerModule.isExported("com.management"); // true  
  
boolean playerModulePnOpen   
 = playerModule.isOpen("com.members"); // false  
boolean managerModulePnOpen   
 = managerModule.isOpen("com.management"); // falseCopy

Beside getting information, the Module class allows us to alter a module. For example, the org.player module doesn't export the com.members package to the org.tournament module. We can check this quickly:

boolean before = playerModule.isExported(  
 "com.members", managerModule); // falseCopy

But we can alter this via reflection. We can perform this export via the Module.addExports() method (in the same category we have addOpens(), addReads(), and addUses()):

playerModule.addExports("com.members", managerModule);Copy

Now, let's check again:

boolean after = playerModule.isExported(  
 "com.members", managerModule); // trueCopy

A module also takes advantages of its own descriptor. The ModuleDescriptor class can be used as the starting point for working with a module:

ModuleDescriptor descriptorPlayerModule   
 = playerModule.getDescriptor();Copy

For example, we can fetch the packages of a module as follows:

Set<String> pcks = descriptorPlayerModule.packages();Copy

165. Dynamic proxies

*Dynamic proxies* can be used to support the implementation of different functionalities that are part of the C**ross Cutting-Concerns** (**CCC**) category. CCC are those concerns that represent ancillary functionalities of the core functionalities, such as database connection management, transaction management (for example, Spring @Transactional), security, and logging.

More precisely, Java Reflection comes with a class named java.lang.reflect.Proxy, the main purpose of which is to provide support for creating dynamic implementations of interfaces at runtime. Proxy reflects on the concrete interface's implementation at runtime.

We can think of Proxy as a *front-wrapper* that pass our invocations to the right methods. Optionally, Proxy can interfere in the process before delegating the invocation.

Dynamic proxies rely on a single class (InvocationHandler) with a single method (invoke()) as in the following diagram:

Diagrama

Descripción generada automáticamente

If we depict the flow from this diagram, then we obtain the following steps:

1. The actors call the needed methods through the exposed *dynamic proxy* (for example, if we want to call the List.add() method, we will do it through a dynamic proxy, not directly)
2. The dynamic proxy will dispatch the invocation to an instance of an InvocationHandler implementation (each proxy instance has an associated invocation handler)
3. The dispatched invocation will hit the invoke() method as a triad containing the proxy object, the method to invoke (as a Method instance) and an array of arguments for this method
4. The InvocationHandler will run additional optional functionalities (for example, CCC) and will invoke the corresponding method
5. The InvocationHandler will return the result of invocation as an object

If we try to resume this flow, then we can say that a dynamic proxy sustains invocations of multiple methods of arbitrary classes via a single class (InvocationHandler) with a single method (invoke()).

Implementing a dynamic proxy

For example, let's write a dynamic proxy that counts the number of invocations of the methods of List.

A dynamic proxy is created via the Proxy.newProxyInstance() method. The newProxyInstance() methods takes three parameters:

* ClassLoader: This is used to load the dynamic proxy class
* Class<?>[]: This is the array of interfaces to implement
* InvocationHandler: This is the invocation handler to dispatch method invocations to

Check out this example:

List<String> listProxy = (List<String>) Proxy.newProxyInstance(  
 List.class.getClassLoader(), new Class[] {  
 List.class}, invocationHandler);Copy

This snippet of code returns a dynamic implementation of the List interface. Further, all invocations via this proxy will be dispatched to the invocationHandler instance.

Mainly, a skeleton of an InvocationHandler implementation looks as follows:

public class DummyInvocationHandler implements InvocationHandler {  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args)  
 throws Throwable {  
 ...  
 }  
}Copy

Since we want to count the number of invocations of the methods of List, we should store all methods signatures and the number of invocations for each of them. This can be accomplished via Map initialized in the constructor of CountingInvocationHandler as follows (this is our InvocationHandler implementation, and invocationHandler is an instance of it):

public class CountingInvocationHandler implements InvocationHandler {  
  
 private final Map<String, Integer> counter = new HashMap<>();  
 private final Object targetObject;  
  
 public CountingInvocationHandler(Object targetObject) {  
 this.targetObject = targetObject;  
  
 for (Method method:targetObject.getClass().getDeclaredMethods()) {  
 this.counter.put(method.getName()   
 + Arrays.toString(method.getParameterTypes()), 0);  
 }  
 }  
 ...  
}Copy

The targetObject field holds the implementation of the List interface (in this case, ArrayList).

And we create a CountingInvocationHandler instance as follows:

CountingInvocationHandler invocationHandler   
 = new CountingInvocationHandler(new ArrayList<>());Copy

The invoke() method simply counts the invocations and invokes Method with the specified arguments:

@Override  
public Object invoke(Object proxy, Method method, Object[] args)  
 throws Throwable {  
  
 Object resultOfInvocation = method.invoke(targetObject, args);  
 counter.computeIfPresent(method.getName()   
 + Arrays.toString(method.getParameterTypes()), (k, v) -> ++v);  
  
 return resultOfInvocation;  
}Copy

Finally, we expose a method that returns the number of invocations for the given method:

public Map<String, Integer> countOf(String methodName) {  
  
 Map<String, Integer> result = counter.entrySet().stream()  
 .filter(e -> e.getKey().startsWith(methodName + "["))  
 .filter(e -> e.getValue() != 0)  
 .collect(Collectors.toMap(Entry::getKey, Entry::getValue));  
  
 return result;  
}Copy

The code bundled to this book glues these snippets of code in a class named CountingInvocationHandler.

At this moment, we can use listProxy to call several methods, as follows:

listProxy.add("Adda");  
listProxy.add("Mark");  
listProxy.add("John");  
listProxy.remove("Adda");  
listProxy.add("Marcel");  
listProxy.remove("Mark");  
listProxy.add(0, "Akiuy");Copy

And let's see how many times we invoked the add() and remove() methods:

// {add[class java.lang.Object]=4, add[int, class java.lang.Object]=1}  
invocationHandler.countOf("add");  
  
// {remove[class java.lang.Object]=2}  
invocationHandler.countOf("remove");Copy

Since the add() method has been invoked via two of its signatures, the resulted Map contains two entries.

Summary

This was the last problem of this chapter. Hopefully, we have finished this comprehensive traversal of the Java Reflection API. We have covered in detail problems regarding classes, interfaces, constructors, methods, fields, annotations, and so on.

Download the applications from this chapter to see the results and to see additional details.

Functional Style Programming - Fundamentals and Design Patterns

This chapter includes 11 problems that involve Java functional-style programming. We will start with a problem that's meant to provide a complete journey from 0 to functional interfaces. Then, we will continue by looking at a suite of design patterns from GoF that we will interpret in Java functional style.

By the end of this chapter, you should be familiar with functional-style programming and ready to continue with a set of problems that allow us to deep dive into this topic. You should be able to use a bunch of commonly used design patterns written in functional-style and have a very good understanding of how to evolve code to take advantage of functional interfaces.

Problems

Use the following problems to test your functional style programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Writing functional interfaces**:Write a program to define the road from 0 to a functional interface via a set of meaningful examples.
2. **Lambdas in a nutshell**:Explain what a lambda expression is.
3. **Implementing the Execute Around pattern**:Write a program that represents an implementation of the Execute Around pattern based on lambdas.
4. **Implementing the Factory pattern**:Write a program that represents an implementation of the Factory pattern based on lambdas.
5. **Implementing the Strategy pattern**:Write a program that represents an implementation of the Strategy pattern based on lambdas.
6. **Implementing the Template Method pattern**:Write a program that represents an implementation of the Template Method pattern based on lambdas.
7. **Implementing the Observer pattern**:Write a program that represents an implementation of the Observer pattern based on lambdas.
8. **Implementing the Loan pattern**:Write a program that represents an implementation of the Loan pattern based on lambdas.
9. **Implementing the Decorator pattern**:Write a program that represents an implementation of the Decorator pattern based on lambdas.
10. **Implementing the Cascaded Builder pattern**:Write a program that represents an implementation of the Cascaded Builder pattern based on lambdas.
11. **Implementing the Command pattern**:Write a program that represents an implementation of the Command pattern based on lambdas.

Solutions

The following sections describe solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations that are shown here only include the most interesting and important details that are needed to solve these problems. You can download the example solutions to view additional details and experiment with the programs from <https://github.com/PacktPublishing/Java-Coding-Problems>.

166. Writing functional interfaces

In this solution, we will highlight the purpose and usability of a functional interface in comparison with several alternatives. We will look at how to evolve the code from its basic and rigid implementation to a flexible implementation based on a functional interface. For this, let's consider the following Melon class:

public class Melon {  
  
 private final String type;  
 private final int weight;  
 private final String origin;  
  
 public Melon(String type, int weight, String origin) {  
 this.type = type;  
 this.weight = weight;  
 this.origin = origin;  
 }  
  
 // getters, toString(), and so on omitted for brevity  
}Copy

Let's assume that we have a client – let's call him Mark – who wants to start up a melon-selling business. We shaped the preceding class based on his description. His main goal is to have an inventory application that will sustain his ideas and decisions, so an application needs to be created that must grow based on business requirements and evolution. We'll take a look at the time that's needed to develop this application on a daily basis in the following sections.

Day 1 (filtering melons by their type)

One day, Mark asked us to provide a feature for filtering melons by their type. As a result, we created a utility class named Filters and implemented a static method that takes a list of melons and the type to filter on as arguments.

The resulting method is pretty straightforward:

public static List<Melon> filterByType(  
 List<Melon> melons, String type) {  
  
 List<Melon> result = new ArrayList<>();  
  
 for (Melon melon: melons) {  
 if (melon != null && type.equalsIgnoreCase(melon.getType())) {  
 result.add(melon);  
 }  
 }  
  
 return result;  
}Copy

Done! Now, we can easily filter melons by type, as shown in the following example:

List<Melon> bailans = Filters.filterByType(melons, "Bailan");Copy

Day 2 (filtering melons of a certain weight)

While Mark was satisfied with the result, he requested another filter to obtain melons of a certain weight (for example, all the melons that are 1,200 grams). We've just implemented a filter like this for melon types, and so we can come up with a new static method for melons of a certain weight, as follows:

public static List<Melon> filterByWeight(  
 List<Melon> melons, int weight) {  
  
 List<Melon> result = new ArrayList<>();  
  
 for (Melon melon: melons) {  
 if (melon != null && melon.getWeight() == weight) {  
 result.add(melon);  
 }  
 }  
  
 return result;  
}Copy

This is similar to filterByType(), except it has a different condition/filter. As developers, we are starting to understand that, if we continue like this, then the Filters class will end up with a lot of methods that simply repeat the code and use a different condition. We are very close to a *boilerplate code* case here.

Day 3 (filtering melons by type and weight)

Things are getting even worse. Mark has now asked us to add a new filter that filters melons by type and weight, and he needs this quickly. However, the quickest implementation is the ugliest. Check it out:

public static List<Melon> filterByTypeAndWeight(  
 List<Melon> melons, String type, int weight) {  
 List<Melon> result = new ArrayList<>();  
  
 for (Melon melon: melons) {  
 if (melon != null && type.equalsIgnoreCase(melon.getType())   
 && melon.getWeight() == weight) {  
 result.add(melon);  
 }  
 }  
  
 return result;  
}Copy

In our context, this is unacceptable. If we add a new filter criterion here, the code will become hard to maintain as well as prone to errors.

Day 4 (pushing the behavior as a parameter)

Meeting time! We cannot continue to add more filters like this; filtering with every attribute we can think of will end up in a huge Filters class that has big, complex methods with too many parameters and tons of *boilerplate* code.

The main problem is that we have different behaviors wrapped in *boilerplate* code. So, it will be nice to write the *boilerplate* code only once and push the behavior as a parameter. This way, we can shape any selection condition/criteria as behavior and juggle them as desired. The code will become more clear, flexible, easy to maintain, and have fewer parameters.

This is known as **Behavior Parameterization**, which is illustrated in the following diagram (the left-hand side shows what we have now; the right-hand side shows what we want):

Forma, Flecha

Descripción generada automáticamente

If we think of each selection condition/criteria as a behavior, then it is pretty intuitive to think of each behavior as an implementation of an interface. Basically, all these behaviors have something in common – a selection condition/criteria and a return of the boolean type (this is known as a **predicate**). In the context of an interface, this is a contract that can be written as follows:

public interface MelonPredicate {  
 boolean test(Melon melon);  
}Copy

Furthermore, we can write different implementations of MelonPredicate. For example, filtering the Gac melons can be written like this:

public class GacMelonPredicate implements MelonPredicate {  
 @Override  
 public boolean test(Melon melon) {  
 return "gac".equalsIgnoreCase(melon.getType());  
 }  
}Copy

Alternatively, filtering all the melons that are heavier than 5,000g can be written:

public class HugeMelonPredicate implements MelonPredicate {  
 @Override  
 public boolean test(Melon melon) {  
 return melon.getWeight() > 5000;  
 }  
}Copy

This technique has a name – the Strategy design pattern. According to GoF, this can "*Define a family of algorithms, encapsulate each one, and make them interchangeable. The strategy pattern lets the algorithm vary independently from client to client.*"

So, the main idea is to dynamically select the behavior of an algorithm at runtime. The MelonPredicate interface unifies all the algorithms dedicated to selecting melons, and each implementation of it is a strategy.

At the moment, we have the strategies, but we don't have any method that receives a MelonPredicate parameter. We need a filterMelons() method, as shown in the following diagram:

Diagrama

Descripción generada automáticamente

So, we need a single parameter and multiple behaviors. Let's look at the source code for filterMelons():

public static List<Melon> filterMelons(  
 List<Melon> melons, MelonPredicate predicate) {  
  
 List<Melon> result = new ArrayList<>();  
  
 for (Melon melon: melons) {  
 if (melon != null && predicate.test(melon)) {  
 result.add(melon);  
 }  
 }  
  
 return result;  
}Copy

This is much better! We can reuse this method with different behaviors as follows (here, we pass GacMelonPredicate and HugeMelonPredicate):

List<Melon> gacs = Filters.filterMelons(  
 melons, new GacMelonPredicate());  
  
List<Melon> huge = Filters.filterMelons(  
 melons, new HugeMelonPredicate());Copy

Day 5 (implementing another 100 filters)

Mark has asked us to implement another 100 filters. This time, we have the flexibility and the support to accomplish this task, but we still need to write 100 strategies or classes for implementing the MelonPredicate for each selection criteria. Moreover, we have to create instances of these strategies and pass them to the filterMelons() method.

This means a lot of code and time. In order to save both, we can rely on Java anonymous classes. In other words, having classes with no names that are declared and instantiated at the same time will result in something like this:

List<Melon> europeans = Filters.filterMelons(  
 melons, new MelonPredicate() {  
 @Override  
 public boolean test(Melon melon) {  
 return "europe".equalsIgnoreCase(melon.getOrigin());  
 }  
});Copy

There is some progress being made here, but this is not very significant because we still need to write a lot of code. Check the highlighted code in the following diagram (this code repeats for each implemented behavior):

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Here, the code is not friendly. Anonymous classes seem complex and they somehow look incomplete and weird, especially to novices.

Day 6 (anonymous classes can be written as lambdas)

A new day, a new idea! Any smart IDE can show us the road ahead. For example, the NetBeans IDE will discretely warn us that this anonymous class can be written as a lambda expression.

This is shown in the following screenshot:

Texto

Descripción generada automáticamente

The message is crystal clear – This anonymous inner class creation can be turned into a lambda expression. Here, make the transformation by hand or let the IDE do it for us.

The result will look like this:

List<Melon> europeansLambda = Filters.filterMelons(  
 melons, m -> "europe".equalsIgnoreCase(m.getOrigin()));Copy

This is much better! Java 8 lambda expressions did a great job this time. Now, we can write Mark's filters in a more flexible, fast, clean, readable, and maintainable manner.

Day 7 (abstracting the List type)

Mark comes up with some good news the next day – he will extend his business and sell other fruits as well as melons. This is cool, but our predicate only supports Melon instances.

So, how should we proceed to support other fruits too? How many other fruits? What if Mark decides to start selling another category of products, such as vegetables? We cannot simply create a predicate for each of them. This will take us back to the start.

The obvious solution is to abstract the List type. We start this by defining a new interface, and this time name it Predicate (remove Melon from the name):

@FunctionalInterface  
public interface Predicate<T> {  
 boolean test(T t);  
}Copy

Next, we rewrite the filterMelons() method and rename it as filter():

public static <T> List<T> filter(  
 List<T> list, Predicate<T> predicate) {  
  
 List<T> result = new ArrayList<>();  
  
 for (T t: list) {  
 if (t != null && predicate.test(t)) {  
 result.add(t);  
 }  
 }  
  
 return result;  
}Copy

Now, we can write filters for Melon:

List<Melon> watermelons = Filters.filter(  
 melons, (Melon m) -> "Watermelon".equalsIgnoreCase(m.getType()));Copy

We can also do the same for numbers:

List<Integer> numbers = Arrays.asList(1, 13, 15, 2, 67);  
List<Integer> smallThan10 = Filters  
 .filter(numbers, (Integer i) -> i < 10);Copy

Take a step back and look at where we started and where we are now. The difference is huge thanks to Java 8 functional interfaces and lambda expressions. Have you noticed the @FunctionalInterface annotation on the Predicate interface? Well, that is an informative annotation type that's used to mark a functional interface. It is useful for an error to occur if the marked interface is not functional.

Conceptually, a functional interface has exactly one abstract method. Moreover, the Predicate interface that we've defined already exists in Java 8 as the java.util.function.Predicate interface. The java.util.function package contains 40+ such interfaces. Consequently, before defining a new one, it is advisable to check this package's content. Most of the time, the six standard built-in functional interfaces will do the job. These are listed as follows:

* Predicate<T>
* Consumer<T>
* **Supplier<T>**
* Function<T, R>
* UnaryOperator<T>
* BinaryOperator<T>

Functional interfaces and lambda expressions make a great team. Lambda expressions support the implementation of the abstract method of a functional interface directly inline. Basically, the entire expression is perceived as an instance of a concrete implementation of the functional interface, as demonstrated in the following code:

Predicate<Melon> predicate = (Melon m)   
 -> "Watermelon".equalsIgnoreCase(m.getType());Copy

167. Lambdas in a nutshell

Dissecting a lambda expression will reveal three main parts, as shown in the following diagram:

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

The following is a description of each part of a lambda expression:

* On the left-hand side of the arrow, we have the parameters of this lambda that are used in the lambda body. These are the parameters of the FilenameFilter.accept​(File folder, String fileName) method.
* On the right-hand of the arrow, we have the lambda body, which in this case checks if the folder in which the file was found can be read and if the file name ends with the .pdf suffix.
* The arrow is just a separator of the lambda's parameters and body.

The anonymous class version of this lambda is as follows:

FilenameFilter filter = new FilenameFilter() {  
 @Override  
 public boolean accept(File folder, String fileName) {  
 return folder.canRead() && fileName.endsWith(".pdf");  
 }  
};Copy

Now, if we look at the lambda and the anonymous version of it, then we can conclude that a lambda expression is a concise anonymous function that can be passed as an argument to a method or kept in a variable. We can conclude that a lambda expression can be described according to the four words shown in the following diagram:

Imagen que contiene Diagrama

Descripción generada automáticamente

Lambdas sustain Behavior Parameterization and that is a big plus (check the previous problem for a detailed explanation of this). Finally, keep in mind that lambdas can be used only in the context of a functional interface.

168. Implementing the Execute Around pattern

The Execute Around pattern tries to eliminate the *boilerplate* code that surrounds specific tasks. For example, the tasks specific to a file need to be surrounded by code for the purpose of opening and closing the file.

Mainly, the Execute Around pattern is useful in scenarios that imply tasks that take place inside a resource open-close lifespan. For example, let's assume that we have a Scanner and that our first task consists of reading a double value from a file:

try (Scanner scanner = new Scanner(  
 Path.of("doubles.txt"), StandardCharsets.UTF\_8)) {  
  
 if (scanner.hasNextDouble()) {  
 double value = scanner.nextDouble();  
 }  
}Copy

Later on, another task consists of printing all double values:

try (Scanner scanner = new Scanner(  
 Path.of("doubles.txt"), StandardCharsets.UTF\_8)) {  
 while (scanner.hasNextDouble()) {  
 System.out.println(scanner.nextDouble());  
 }  
}Copy

The following diagram highlights the *boilerplate* code that surrounds these two tasks:

Texto

Descripción generada automáticamente

In order to avoid this *boilerplate* code, the Execute Around pattern relies on Behavior Parameterization (further detailed in the *Writing functional interfaces* section). The steps that are needed to accomplish this are as follows:

1. The first step is to define a functional interface that matches the Scanner -> double signature, which may throw an IOException:

@FunctionalInterface  
public interface ScannerDoubleFunction {  
 double readDouble(Scanner scanner) throws IOException;  
}Copy

Declaring the functional interface is just half of the solution.

1. So far, we can write a lambda of the Scanner -> double type, but we need a method that receives it and executes it. For this, let's consider the following method in the Doubles utility class:

public static double read(ScannerDoubleFunction snf)  
 throws IOException {  
  
 try (Scanner scanner = new Scanner(  
 Path.of("doubles.txt"), StandardCharsets.UTF\_8)) {  
  
 return snf.readDouble(scanner);  
 }  
}Copy

The lambda that's passed to the read() method is executed inside the body of this method. When we pass the lambda, we provide an implementation of the abstract method known as readDouble() directly inline. Mainly, this is perceived as an instance of the functional interface, ScannerDoubleFunction, and so we can call the readDouble() method to obtain the desired result.

1. Now, we can simply pass our tasks as lambdas and reuse the read() method. For example, our tasks can be wrapped in two static methods, as shown here (this practice is needed to obtain clean code and avoid big lambdas):

private static double getFirst(Scanner scanner) {  
 if (scanner.hasNextDouble()) {  
 return scanner.nextDouble();  
 }  
  
 return Double.NaN;  
}  
  
private static double sumAll(Scanner scanner) {  
 double sum = 0.0d;  
 while (scanner.hasNextDouble()) {  
  
 sum += scanner.nextDouble();  
 }  
  
 return sum;  
}Copy

1. Having these two tasks as examples, we can write other tasks as well. Let's pass them to the read() method:

double singleDouble   
 = Doubles.read((Scanner sc) -> getFirst(sc));  
double sumAllDoubles   
 = Doubles.read((Scanner sc) -> sumAll(sc));Copy

The Execute Around pattern is quite useful for eliminating the *boilerplate* code that's specific for opening and closing resources (I/O operations).

169. Implementing the Factory pattern

In a nutshell, the Factory pattern allows us to create several kinds of objects without exposing the instantiation process to the caller. This way, we can hide the complex and/or sensitive process of creating objects and expose an intuitive and easy-to-use factory of objects to the caller.

In a classic implementation, the Factory pattern relies on an intern switch(), as shown in the following example:

public static Fruit newInstance(Class<?> clazz) {  
 switch (clazz.getSimpleName()) {  
 case "Gac":  
 return new Gac();  
 case "Hemi":  
 return new Hemi();  
 case "Cantaloupe":  
 return new Cantaloupe();  
 default:  
 throw new IllegalArgumentException(  
 "Invalid clazz argument: " + clazz);  
 }  
}Copy

Here, Gac, Hemi, and Cantaloupe are implementing the same Fruit interface and have an empty constructor. If this method lives in a utility class named MelonFactory, we can call it as follows:

Gac gac = (Gac) MelonFactory.newInstance(Gac.class);Copy

However, Java 8 functional-style allows us to refer to constructors using the *method references* technique. This means that we can define a Supplier<Fruit> to refer to the Gac empty constructor, as follows:

Supplier<Fruit> gac = Gac::new;Copy

How about Hemi, Cantaloupe, and so on? Well, we can simply put all of them in a Map (notice that no melon type is instantiated here; these are just lazy *method references*):

private static final Map<String, Supplier<Fruit>> MELONS   
 = Map.of("Gac", Gac::new, "Hemi", Hemi::new,  
 "Cantaloupe", Cantaloupe::new);Copy

Furthermore, we can rewrite the newInstance() method to use this map:

public static Fruit newInstance(Class<?> clazz) {  
  
 Supplier<Fruit> supplier = MELONS.get(clazz.getSimpleName());  
  
 if (supplier == null) {  
 throw new IllegalArgumentException(  
 "Invalid clazz argument: " + clazz);  
 }  
  
 return supplier.get();  
 }Copy

The caller code doesn't need any further modifications:

Gac gac = (Gac) MelonFactory.newInstance(Gac.class);Copy

However, obviously, constructors are not always empty. For example, the following Melon class exposes a single constructor with three arguments:

public class Melon implements Fruit {  
  
 private final String type;  
 private final int weight;  
 private final String color;  
  
 public Melon(String type, int weight, String color) {  
 this.type = type;  
 this.weight = weight;  
 this.color = color;  
 }  
}Copy

Creating an instance of this class cannot be obtained via an empty constructor. But if we define a functional interface that supports three arguments and a return, then we are back on track:

@FunctionalInterface  
public interface TriFunction<T, U, V, R> {  
 R apply(T t, U u, V v);  
}Copy

This time, the following statement will try to fetch a constructor with three arguments of the String, Integer, and String types:

private static final  
 TriFunction<String, Integer, String, Melon> MELON = Melon::new;Copy

The newInstance() method, which was made especially for the Melon class is:

public static Fruit newInstance(  
 String name, int weight, String color) {  
 return MELON.apply(name, weight, name);  
}Copy

A Melon instance can be created as follows:

Melon melon = (Melon) MelonFactory.newInstance("Gac", 2000, "red");Copy

Done! Now, we have a factory of Melon via functional interfaces.

170. Implementing the Strategy pattern

The classic Strategy pattern is pretty straightforward. It consists of an interface that represents a family of algorithms (strategies) and several implementations of this interface (each implementation is a strategy).

For example, the following interface unifies the strategies for removing characters from the given string:

public interface RemoveStrategy {  
 String execute(String s);  
}Copy

First, we will define a strategy for removing numeric values from a string:

public class NumberRemover implements RemoveStrategy {  
 @Override  
 public String execute(String s) {  
 return s.replaceAll("\\d", "");  
 }  
}Copy

Then, we will define a strategy for removing white spaces from a string:

public class WhitespacesRemover implements RemoveStrategy {  
 @Override  
 public String execute(String s) {  
 return s.replaceAll("\\s", "");  
 }  
}Copy

Finally, let's define a utility class that acts as the entry point for strategies:

public final class Remover {  
  
 private Remover() {  
 throw new AssertionError("Cannot be instantiated");  
 }  
  
 public static String remove(String s, RemoveStrategy strategy) {  
 return strategy.execute(s);  
 }  
}Copy

This is a simple and classical Strategy pattern implementation. If we want to remove numeric values from a string, we can do this as follows:

String text = "This is a text from 20 April 2050";  
String noNr = Remover.remove(text, new NumberRemover());Copy

But do we actually need the NumberRemover and WhitespacesRemover classes? Do we need to write similar classes for further strategies? Obviously, the answer is no.

Check out our interface one more time:

@FunctionalInterface  
public interface RemoveStrategy {  
 String execute(String s);  
}Copy

We've just added the @FunctionalInterface hint because the RemoveStrategy interface defines a single abstract method, and so it is a functional interface.

What can we use in the context of a functional interface? Well, the obvious answer is lambdas. Moreover, what can a lambda do for us in this scenario? It can remove the *boilerplate* code (in this case, the classes representing the strategies) and encapsulate the strategy in its body:

String noNr = Remover.remove(text, s -> s.replaceAll("\\d", ""));  
String noWs = Remover.remove(text, s -> s.replaceAll("\\s", ""));Copy

So, this is what the Strategy pattern looks like via lambdas.

171. Implementing the Template Method pattern

The Template Method is a classical design pattern from GoF that allows us to write a skeleton of an algorithm in a method and defer certain steps of this algorithm to the client subclasses.

For example, making a pizza involves three main steps – preparing the dough, adding toppings, and baking the pizza. While the first and last step can be considered the same (fixed steps) for all pizzas, the second step is different for each type of pizza (variable step).

If we put this in code via the Template Method pattern, then we obtain something like the following (the make() method represents the template method and contains the fixed and variable steps in a well-defined order):

public abstract class PizzaMaker {  
  
 public void make(Pizza pizza) {  
 makeDough(pizza);  
 addTopIngredients(pizza);  
 bake(pizza);  
 }  
  
 private void makeDough(Pizza pizza) {  
 System.out.println("Make dough");  
 }  
  
 private void bake(Pizza pizza) {  
 System.out.println("Bake the pizza");  
 }  
  
 public abstract void addTopIngredients(Pizza pizza);  
}Copy

The fixed steps have default implementations while the variable step is represented by an abstract method called addTopIngredients(). This method is implemented by subclasses of this class. For example, a Neapolitan pizza will be abstracted as follows:

public class NeapolitanPizza extends PizzaMaker {  
  
 @Override  
 public void addTopIngredients(Pizza p) {  
 System.out.println("Add: fresh mozzarella, tomatoes,  
 basil leaves, oregano, and olive oil ");  
 }  
}Copy

On the other hand, a Greek pizza will be as follows:

public class GreekPizza extends PizzaMaker {  
  
 @Override  
 public void addTopIngredients(Pizza p) {  
 System.out.println("Add: sauce and cheese");  
 }  
}Copy

So, each type of pizza requires a new class that overrides the addTopIngredients() method. In the end, we can make a pizza like so:

Pizza nPizza = new Pizza();  
PizzaMaker nMaker = new NeapolitanPizza();  
nMaker.make(nPizza);Copy

The drawback of this approach consists of *boilerplate* code and verbosity. However, we can tackle this drawback via lambdas. We can represent the variable steps of the Template Method as lambdas expressions. Depending on the case, we have to choose the proper functional interfaces. In our case, we can rely on a Consumer, as follows:

public class PizzaLambda {  
  
 public void make(Pizza pizza, Consumer<Pizza> addTopIngredients) {  
 makeDough(pizza);  
 addTopIngredients.accept(pizza);  
 bake(pizza);  
 }  
  
 private void makeDough(Pizza p) {  
 System.out.println("Make dough");  
 }  
  
 private void bake(Pizza p) {  
 System.out.println("Bake the pizza");  
 }  
}Copy

This time, there is no need to define subclasses (no need to have NeapolitanPizza, GreekPizza, or others). We just pass the variable step via a lambda expression. Let's make a Sicilian pizza:

Pizza sPizza = new Pizza();  
new PizzaLambda().make(sPizza, (Pizza p)   
 -> System.out.println("Add: bits of tomato, onion,  
 anchovies, and herbs "));Copy

Done! No more *boilerplate* code is needed. The lambda solution has seriously improved the solution.

172. Implementing the Observer pattern

In a nutshell, the Observer pattern relies on an object (known as the **subject**) that automatically notifies its subscribers (known as **observers**) when certain events happen.

For example, the fire station headquarters can be the *subject*, and the local fire stations can be the *observers*. When a fire has started, the fire station headquarters notifies all local fire stations and sends them the address where the fire is taking place. Each *observer* analyzes the received address and, depending on different criteria, decides whether to extinguish the fire or not.

All the local fire stations are grouped via an interface called FireObserver. This method defines a single abstract method that is invoked by the fire station headquarters (*subject*):

public interface FireObserver {  
 void fire(String address);  
}Copy

Each local fire station (*observer*) implements this interface and decides whether to extinguish the fire or not in the fire() implementation. Here, we have three local stations (Brookhaven, Vinings, and Decatur):

public class BrookhavenFireStation implements FireObserver {  
  
 @Override  
 public void fire(String address) {  
 if (address.contains("Brookhaven")) {  
 System.out.println(  
 "Brookhaven fire station will go to this fire");  
 }  
 }  
}  
  
public class ViningsFireStation implements FireObserver {  
 // same code as above for ViningsFireStation  
}  
  
public class DecaturFireStation implements FireObserver {  
 // same code as above for DecaturFireStation  
}Copy

Half of the job is done! Now, we need to register these *observers* to be notified by the *subject*. In other words, each local fire station needs to be registered as an *observer* to the fire station headquarters (*subject*). For this, we declare another interface that defines the *subject* contract for registering and notifying its *observers*:

public interface FireStationRegister {  
 void registerFireStation(FireObserver fo);  
 void notifyFireStations(String address);  
}Copy

Finally, we can write the fire station headquarters (*subject*):

public class FireStation implements FireStationRegister {  
  
 private final List<FireObserver> fireObservers = new ArrayList<>();  
  
 @Override  
 public void registerFireStation(FireObserver fo) {  
 if (fo != null) {  
 fireObservers.add(fo);  
 }  
 }  
  
 @Override  
 public void notifyFireStations(String address) {  
 if (address != null) {  
 for (FireObserver fireObserver: fireObservers) {  
 fireObserver.fire(address);  
 }  
 }  
 }  
}Copy

Now, let's register our three local stations (*observers*) to the fire station headquarters (*subject*):

FireStation fireStation = new FireStation();  
fireStation.registerFireStation(new BrookhavenFireStation());  
fireStation.registerFireStation(new DecaturFireStation());  
fireStation.registerFireStation(new ViningsFireStation());Copy

Now, when a fire occurs, the fire station headquarters will notify all registered local fire stations:

fireStation.notifyFireStations(  
 "Fire alert: WestHaven At Vinings 5901 Suffex Green Ln Atlanta");Copy

The Observer pattern was successfully implemented there.

This is another classical case of *boilerplate* code. Each local fire station needs a new class and implementation of the fire() method.

However, lambdas can help us again! Check out the FireObserver interface. It has a single abstract method; therefore, this is a functional interface:

@FunctionalInterface  
public interface FireObserver {  
 void fire(String address);  
}Copy

This functional interface is an argument of the Fire.registerFireStation() method. In this context, we can pass a lambda to this method instead of a new instance of a local fire station. The lambda will contain the behavior in its body; therefore, we can delete the local station classes and rely on lambdas, as follows:

fireStation.registerFireStation((String address) -> {  
 if (address.contains("Brookhaven")) {  
 System.out.println(  
 "Brookhaven fire station will go to this fire");  
 }  
});  
  
fireStation.registerFireStation((String address) -> {  
 if (address.contains("Vinings")) {  
 System.out.println("Vinings fire station will go to this fire");  
 }  
});  
  
fireStation.registerFireStation((String address) -> {  
 if (address.contains("Decatur")) {  
 System.out.println("Decatur fire station will go to this fire");  
 }  
});Copy

Done! No more *boilerplate* code.

173. Implementing the Loan pattern

In this problem, we will talk about implementing the Loan pattern. Let's assume that we have a file containing three numbers (let's say, doubles), and each number is a coefficient of a formula. For example, the numbers *x*, *y*, and *z* are the coefficients of the following two formulas: *x+y-z* and *x-y\*sqrt(z).* In the same manner, we can write other formulas as well.

At this point, we have enough experience to recognize that this scenario sounds like a good fit for Behavior Parameterization. This time, we don't define a custom functional interface, and we use a built-in functional interface called Function<T, R>. This functional interface represents a function that accepts one argument and produces a result. The signature of its abstract method is R apply(T t).

This functional interface becomes an argument of a static method that's meant to implement the Loan pattern. Let's place this method in a class called Formula:

public class Formula {  
 ...  
 public static double compute(  
 Function<Formula, Double> f) throws IOException {  
 ...  
 }  
}Copy

Notice that the compute() method accepts lambdas of the Formula -> Double type while it is declared in the Formula class. Let's reveal the entire source code of compute():

public static double compute(  
 Function<Formula, Double> f) throws IOException {  
  
 Formula formula = new Formula();  
 double result = 0.0 d;  
  
 try {  
 result = f.apply(formula);  
 } finally {  
 formula.close();  
 }  
  
 return result;  
}Copy

There are three points that should be highlighted here. First, when we create a new instance of Formula, we actually open a new Scanner to our file (check the private constructor of this class):

public class Formula {  
  
 private final Scanner scanner;  
 private double result;  
  
 private Formula() throws IOException {  
 result = 0.0 d;  
  
 scanner = new Scanner(  
 Path.of("doubles.txt"), StandardCharsets.UTF\_8);  
 }  
 ...  
}Copy

Second, when we execute the lambda, we are actually calling a chain of instance methods of Formula that perform the computation (apply the formula). Each of these methods returns the current instance. The instance methods that should be called are defined in the body of the lambda expression.

We only need the following computations, but more can be added:

public Formula add() {  
 if (scanner.hasNextDouble()) {  
 result += scanner.nextDouble();  
 }  
  
 return this;  
}  
  
public Formula minus() {  
 if (scanner.hasNextDouble()) {  
 result -= scanner.nextDouble();  
 }  
  
 return this;  
}  
  
public Formula multiplyWithSqrt() {  
 if (scanner.hasNextDouble()) {  
 result \*= Math.sqrt(scanner.nextDouble());  
 }  
  
 return this;  
}Copy

Since the result of the computation (the formula) is a double, we need to provide a Terminal method that returns the final result:

public double result() {  
 return result;  
}Copy

Finally, we close the Scanner and reset the result. This takes place in the private close() method:

private void close() {  
 try (scanner) {  
 result = 0.0 d;  
 }  
}Copy

These pieces have been glued into the code bundled with this book under a class named Formula.

Now, do you remember our formulas? We had *x+y-z* and *x-y\*sqrt(z)*. The first one can be written as follows:

double xPlusYMinusZ = Formula.compute((sc)  
 -> sc.add().add().minus().result());Copy

The second formula can be written as follows:

double xMinusYMultiplySqrtZ = Formula.compute((sc)  
 -> sc.add().minus().multiplyWithSqrt().result());Copy

Notice that we can focus on our formulas and we don't need to bother with opening and closing the file. Moreover, the fluent API allows us to shape any formula and it is very easy to enrich it with more operations.

174. Implementing the Decorator pattern

The Decorator pattern prefers composition over inheritance; therefore, it is an elegant alternative to the subclassing technique. With this, we mainly start from a base object and add additional features in a dynamic fashion.

For example, we can use this pattern to decorate a cake. The decoration process doesn't change the cake itself – it just adds some nuts, cream, fruit, and so on.

The following diagram illustrates what we will implement:

Diagrama

Descripción generada automáticamente

First, we create an interface called Cake:

public interface Cake {  
 String decorate();  
}Copy

Then, we implement this interface via BaseCake:

public class BaseCake implements Cake {  
  
 @Override  
 public String decorate() {  
 return "Base cake ";  
 }  
}Copy

Afterward, we create an abstract CakeDecorator class for this Cake. The main goal of this class is to call the decorate() method of the given Cake:

public class CakeDecorator implements Cake {  
  
 private final Cake cake;  
  
 public CakeDecorator(Cake cake) {  
 this.cake = cake;  
 }  
  
 @Override  
 public String decorate() {  
 return cake.decorate();  
 }  
}Copy

Next, we focus on writing our decorators.

Each decorator extends CakeDecorator and alters the decorate() method to add the corresponding decoration.

For example, the Nuts decorator looks like this:

public class Nuts extends CakeDecorator {  
  
 public Nuts(Cake cake) {  
 super(cake);  
 }  
  
 @Override  
 public String decorate() {  
 return super.decorate() + decorateWithNuts();  
 }  
  
 private String decorateWithNuts() {  
 return "with Nuts ";  
 }  
}Copy

For brevity purposes, we skip the Cream decorator. However, it is pretty straightforward to intuit that this decorator is mostly the same as Nuts.

So, again, we have some *boilerplate* code.

Now, we can create a Cake decorated with nuts and cream, as follows:

Cake cake = new Nuts(new Cream(new BaseCake()));  
// Base cake with Cream with Nuts  
  
System.out.println(cake.decorate());Copy

So, this is a classical implementation of the Decorator pattern. Now, let's take a look at the lambda-based implementation, which drastically reduces this code. This is especially the case when we have a significant number of decorators.

This time, we transform the Cake interface into a class, as follows:

public class Cake {  
  
 private final String decorations;  
  
 public Cake(String decorations) {  
 this.decorations = decorations;  
 }  
  
 public Cake decorate(String decoration) {  
 return new Cake(getDecorations() + decoration);  
 }  
  
 public String getDecorations() {  
 return decorations;  
 }  
}Copy

The climax here is the decorate() method. Mainly, this method applies the given decoration next to the existing decorations and returns a new Cake.

As another example, let's consider the java.awt.Color class, which has a method named brighter(). This method creates a new Color that is a brighter version of the current Color. Similarly, the decorate() method creates a new Cake that is a more decorated version of the current Cake.

Furthermore, there is no need to write decorators as separate classes. We will rely on lambdas to pass the decorators to the CakeDecorator:

public class CakeDecorator {  
  
 private Function<Cake, Cake> decorator;  
  
 public CakeDecorator(Function<Cake, Cake>... decorations) {  
 reduceDecorations(decorations);  
 }  
  
 public Cake decorate(Cake cake) {  
 return decorator.apply(cake);  
 }  
  
 private void reduceDecorations(  
 Function<Cake, Cake>... decorations) {  
  
 decorator = Stream.of(decorations)  
 .reduce(Function.identity(), Function::andThen);  
 }  
}Copy

Mainly, this class accomplishes two things:

* In the constructor, it calls the reduceDecorations() method. This method will chain the array of the passed Function via the Stream.reduce() and Function.andThen() methods. The result is a single Function composed from the array of the given Function.
* When the apply() method of the composed Function is called from the decorate() method, it will apply the chain of given functions one by one. Since each Function in the given array is a decorator, the composed Function will apply each decorator one by one.

Let's create a Cake decorated with nuts and cream:

CakeDecorator nutsAndCream = new CakeDecorator(  
 (Cake c) -> c.decorate(" with Nuts"),  
 (Cake c) -> c.decorate(" with Cream"));  
  
Cake cake = nutsAndCream.decorate(new Cake("Base cake"));  
  
// Base cake with Nuts with Cream  
System.out.println(cake.getDecorations());Copy

Done! Consider running the code bundled with this book to check the output.

175. Implementing the Cascaded Builder pattern

We already talked about this pattern in [Chapter 2](https://subscription.packtpub.com/book/programming/9781789801415/8/ch08lvl1sec32/16027e66-3bab-4bff-b106-7cbe15480be6.xhtml), *Objects, Immutability, and Switch Expressions*, in the *Writing an immutable class via the Builder pattern* section. It is advisable to treat this problem, just as a quick reminder of the Builder pattern.

Having the classic Builder under our tool belt, let's suppose that we want to write a class for delivering parcels. Mainly, we want to set the receiver's first name, last name, address, and parcel content and then deliver the parcel.

We can accomplish this via the Builder pattern and lambdas, as follows:

public final class Delivery {  
  
 public Delivery firstname(String firstname) {  
 System.out.println(firstname);  
  
 return this;  
 }  
  
 //similar for lastname, address and content  
  
 public static void deliver(Consumer<Delivery> parcel) {  
 Delivery delivery = new Delivery();  
 parcel.accept(delivery);  
  
 System.out.println("\nDone ...");  
 }  
}Copy

For delivering a parcel, we simply use a lambda:

Delivery.deliver(d -> d.firstname("Mark")  
 .lastname("Kyilt")  
 .address("25 Street, New York")  
 .content("10 books"));Copy

Obviously, using lambdas is facilitated by the Consumer<Delivery> argument.

176. Implementing the Command pattern

In a nutshell, the Command pattern is used in scenarios where a command is wrapped in an object. This object can be passed around without being aware of the command itself or the receiver of the command.

A classic implementation of this pattern consists of several classes. In our scenario, we have the following:

* The Command interface is responsible for executing a certain action (in this case, the possible actions are move, copy, and delete). The concrete implementations of this interface are CopyCommand, MoveCommand, and DeleteCommand.
* The IODevice interface defines the supported actions (move(), copy(), and delete()). The HardDisk class is a concrete implementation of IODevice and represents the *receiver*.
* The Sequence class is the *invoker* of the commands, and it knows how to execute a given command. The *invoker* can act in different ways, but in this case, we simply record the commands and execute them in a batch when the runSequence() is called.

The Command pattern can be represented by the following diagram:

Diagrama

Descripción generada automáticamente

So, the HardDisk implements the actions that are given in the IODevice interface. As a *receiver*, the HardDisk is responsible for running the actual action when the execute() method of a certain command is called. The source code for IODevice is as follows:

public interface IODevice {  
 void copy();  
 void delete();  
 void move();  
}Copy

The HardDisk is the concrete implementation of IODevice:

public class HardDisk implements IODevice {  
  
 @Override  
 public void copy() {  
 System.out.println("Copying ...");  
 }  
  
 @Override  
 public void delete() {  
 System.out.println("Deleting ...");  
 }  
  
 @Override  
 public void move() {  
 System.out.println("Moving ...");  
 }  
}Copy

All concrete command classes implement the Command interface:

public interface Command {  
 public void execute();  
}  
  
public class DeleteCommand implements Command {  
  
 private final IODevice action;  
  
 public DeleteCommand(IODevice action) {  
 this.action = action;  
 }  
  
 @Override  
 public void execute() {  
 action.delete()  
 }  
}Copy

In the same manner, we have implemented CopyCommand and MoveCommand and skipped these for brevity purposes.

Furthermore, the Sequence class acts as the *invoker* class. The *invoker* knows how to execute the given command, but it doesn't have any clue about the command's implementation (it only knows the command's interface). Here, we record the commands in a List and execute those commands in a batch when the runSequence() method is called:

public class Sequence {  
  
 private final List<Command> commands = new ArrayList<>();  
  
 public void recordSequence(Command cmd) {  
 commands.add(cmd);  
 }  
  
 public void runSequence() {  
 commands.forEach(Command::execute);  
 }  
  
 public void clearSequence() {  
 commands.clear();  
 }  
}Copy

Now, let's see it at work. Let's execute a batch of actions on HardDisk:

HardDisk hd = new HardDisk();  
Sequence sequence = new Sequence();  
sequence.recordSequence(new CopyCommand(hd));  
sequence.recordSequence(new DeleteCommand(hd));  
sequence.recordSequence(new MoveCommand(hd));  
sequence.recordSequence(new DeleteCommand(hd));  
sequence.runSequence();Copy

Obviously, we have a lot of *boilerplate* code here. Check out the classes of commands. Do we actually need all of these classes? Well, if we realize that the Command interface is actually a functional interface, then we can remove its implementations and provide the behaviors via lambdas (the command classes are just blocks of behavior, and so they can be expressed via lambdas), as follows:

HardDisk hd = new HardDisk();  
Sequence sequence = new Sequence();  
sequence.recordSequence(hd::copy);  
sequence.recordSequence(hd::delete);  
sequence.recordSequence(hd::move);  
sequence.recordSequence(hd::delete);  
sequence.runSequence();Copy

Summary

We have now reached the end of this chapter. Using lambdas to reduce or even eliminate the *boilerplate* code is a technique that can be used in other design patterns and scenarios as well. Having the knowledge you've accumulated so far should provide you with a solid base for adapting your cases accordingly.

Download the applications from this chapter to view the results and additional details.

Functional Style Programming - a Deep Dive

This chapter includes 22 problems that involve Java functional-style programming. Here, we will focus on several problems that involve classical operations that are encountered in streams (such as, filter and map), and discuss infinite streams, null-safe streams, and default methods. This comprehensive list of problems will cover grouping, partitioning, and collectors, including the JDK 12 teeing() collector and writing a custom collector. In addition, takeWhile(), dropWhile(), composing functions, predicates and comparators, testing and debugging lambdas, and other cool topics will be discussed as well.

Once you've covered this and the previous chapter, you will be ready to unleash functional-style programming on your production applications. The following problems will prepare you for a wide range of use cases, including corner cases or pitfalls.

Problems

Use the following problems to test your functional style programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Testing high-order functions**: Write several unit tests for testing so-called high-order functions.
2. **Testing methods that use lambdas**: Write several unit tests for testing methods that use lambdas.
3. **Debugging lambdas**: Provide a technique for debugging lambdas.
4. **Filtering the non-zero elements of a stream**: Write a stream pipeline that filters the non-zero elements of a stream.
5. **Infinite streams,** takeWhile(), **and** dropWhile(): Write several snippets of code that work with infinite streams. In addition, write several examples of working with the takeWhile() and dropWhile() APIs.
6. **Mapping a stream**: Write several examples of mapping a stream via map() and flatMap().
7. **Finding different elements in a stream**: Write a program for finding different elements in a stream.
8. **Matching different elements in a stream**: Write a program for matching different elements in a stream.
9. **Sum, max, and min in a stream**: Write a program for computing the sum, max, and min of the given stream via primitive specializations of Stream and Stream.reduce().
10. **Collecting the results of a stream**: Write several snippets of code for collecting the results of a stream in a list, map, and set.
11. **Joining the results of a stream**: Write several snippets of code for joining the results of a stream into a String.
12. **Summarization collectors**: Write several snippets of code to reveal the usage of summarization collectors.
13. **Grouping**: Write snippets of code for working with groupingBy() collectors.
14. **Partitioning**: Write several snippets of code for working with partitioningBy() collectors.
15. **Filtering, flattening, and mapping collectors**: Write several snippets of code for exemplifying the usage of filtering, flattening, and mapping collectors.
16. **Teeing**: Write several examples that merge the results of two collectors (JDK 12 and Collectors.teeing()).
17. **Writing a custom collector**: Write a program that represents a custom collector.
18. **Method reference**:Write an example of method reference.
19. **Parallel processing of streams**: Provide a brief overview of the parallel processing of streams. Provide at least one example each for parallelStream(), parallel(), and spliterator().
20. **Null-safe streams**: Write a program that returns a null-safe stream from an element or a collection of elements.
21. **Composing functions, predicates, and comparators**: Write several examples for composing functions, predicates, and comparators.
22. **Default methods**:Write an interface that contains a default method.

Solutions

The following sections describe the solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations shown here only include the most interesting and important details that are needed to solve the problems. You can download the example solutions to view additional details and experiment with the programs from <https://github.com/PacktPublishing/Java-Coding-Problems>.

177. Testing high-order functions

A *high-order function* is a term that's used to characterize a function that returns a function or takes a function as a parameter.

Based on this statement, testing a high-order function in the context of lambdas should cover two main cases:

* Testing a method that takes a lambda as a parameter
* Testing a method that returns a functional interface

We'll learn about these two tests in the upcoming sections.

Testing a method that takes a lambda as a parameter

Testing a method that takes a lambda as a parameter can be accomplished by passing different lambdas to this method. For example, let's assume that we have the following functional interface:

@FunctionalInterface  
public interface Replacer<String> {  
 String replace(String s);  
}Copy

Let's also assume that we have a method that takes lambdas of the String -> String type, as follows:

public static List<String> replace(  
 List<String> list, Replacer<String> r) {  
  
 List<String> result = new ArrayList<>();  
 for (String s: list) {  
 result.add(r.replace(s));  
 }  
  
 return result;  
}Copy

Now, let's write a JUnit test for this method using two lambdas:

@Test  
public void testReplacer() throws Exception {  
  
 List<String> names = Arrays.asList(  
 "Ann a 15", "Mir el 28", "D oru 33");  
  
 List<String> resultWs = replace(  
 names, (String s) -> s.replaceAll("\\s", ""));  
 List<String> resultNr = replace(  
 names, (String s) -> s.replaceAll("\\d", ""));  
  
 assertEquals(Arrays.asList(  
 "Anna15", "Mirel28", "Doru33"), resultWs);  
 assertEquals(Arrays.asList(  
 "Ann a ", "Mir el ", "D oru "), resultNr);  
}Copy

Testing a method that returns a functional interface

On the other hand, testing a method that returns a functional interface can be interpreted as testing the behavior of that functional interface. Let's consider the following method:

public static Function<String, String> reduceStrings(  
 Function<String, String> ...functions) {  
  
 Function<String, String> function = Stream.of(functions)  
 .reduce(Function.identity(), Function::andThen);  
  
 return function;  
}Copy

Now, we can test the behavior of the returned Function<String, String> as follows:

@Test  
public void testReduceStrings() throws Exception {  
  
 Function<String, String> f1 = (String s) -> s.toUpperCase();  
 Function<String, String> f2 = (String s) -> s.concat(" DONE");  
  
 Function<String, String> f = reduceStrings(f1, f2);  
  
 assertEquals("TEST DONE", f.apply("test"));  
}Copy

178. Testing methods that use lambdas

Let's start by testing a lambda that is not wrapped in a method. For example, the following lambda is associated with a field (for being reused), and we want to test its logic:

public static final Function<String, String> firstAndLastChar  
 = (String s) -> String.valueOf(s.charAt(0))  
 + String.valueOf(s.charAt(s.length() - 1));Copy

Let's take into account that a lambda generates an instance of a functional interface; then, we can test the behavior of that instance as follows:

@Test  
public void testFirstAndLastChar() throws Exception {  
  
 String text = "Lambda";  
 String result = firstAndLastChar.apply(text);  
 assertEquals("La", result);  
}Copy

Another solution consists of wrapping the lambda in a method call and writing unit tests for the method call.

Often, the lambdas are used inside methods. For most cases, testing the method that contains the lambda is acceptable, but there are cases when we want to test the lambda itself. A solution to this problem consists of three main steps:

1. Extracting the lambda in a static method
2. Replacing the lambda with a *method reference*
3. Testing this static method

For example, let's consider the following method:

public List<String> rndStringFromStrings(List<String> strs) {  
  
 return strs.stream()  
 .map(str -> {  
 Random rnd = new Random();  
 int nr = rnd.nextInt(str.length());  
 String ch = String.valueOf(str.charAt(nr));  
  
 return ch;  
 })  
 .collect(Collectors.toList());  
}Copy

Our goal is to test the lambda from this method:

str -> {  
 Random rnd = new Random();  
 int nr = rnd.nextInt(str.length());  
 String ch = String.valueOf(str.charAt(nr));  
  
 return ch;  
})Copy

So, let's apply the preceding three steps:

1. Let's extract this lambda in a static method:

public static String extractCharacter(String str) {  
  
 Random rnd = new Random();  
 int nr = rnd.nextInt(str.length());  
 String chAsStr = String.valueOf(str.charAt(nr));  
  
 return chAsStr;  
}Copy

1. Let's replace the lambda with the corresponding method reference:

public List<String> rndStringFromStrings(List<String> strs) {  
  
 return strs.stream()  
 .map(StringOperations::extractCharacter)  
 .collect(Collectors.toList());  
}Copy

1. Let's test the static method (which is the lambda):

@Test  
public void testRndStringFromStrings() throws Exception {  
  
 String str1 = "Some";  
 String str2 = "random";  
 String str3 = "text";  
  
 String result1 = extractCharacter(str1);  
 String result2 = extractCharacter(str2);  
 String result3 = extractCharacter(str3);  
  
 assertEquals(result1.length(), 1);  
 assertEquals(result2.length(), 1);  
 assertEquals(result3.length(), 1);  
 assertThat(str1, containsString(result1));  
 assertThat(str2, containsString(result2));  
 assertThat(str3, containsString(result3));  
}Copy

It is advisable to avoid lambdas that have more than one line of code. Therefore, by following the preceding technique, the lambdas become easy to test.

179. Debugging lambdas

There are at least three solutions when it comes to debugging lambdas:

* Inspect a stack trace
* Logging
* Rely on IDE support (for example, NetBeans, Eclipse, and IntelliJ IDEA support debugging lambdas *out of the box* or provide plugins for it)

Let's focus on the first two since relying on an IDE is a very large and specific topic that isn't in the scope of this book.

Inspecting the stack trace of a failure that happened inside a lambda or a stream pipeline can be pretty puzzling. Let's consider the following snippet of code:

List<String> names = Arrays.asList("anna", "bob", null, "mary");  
  
names.stream()  
 .map(s -> s.toUpperCase())  
 .collect(Collectors.toList());Copy

Since the third element from this list is null, we will get a NullPointerException, and the whole sequence of calls that defines the stream pipeline is exposed, as in the following screenshot:

Texto

Descripción generada automáticamente

The highlighted line tells us that this NullPointerException has occurred inside a lambda expression named lambda$main$5. This name was made up by the compiler since lambdas don't have names. Moreover, we don't know which element was null.

So, we can conclude that a stack trace that reports a failure inside a lambda or stream pipeline is not very intuitive.

Alternatively, we can try to log the output. This will help us debug a pipeline of operations in a stream. This can be accomplished via the forEach() method:

List<String> list = List.of("anna", "bob",  
 "christian", "carmen", "rick", "carla");  
  
list.stream()  
 .filter(s -> s.startsWith("c"))  
 .map(String::toUpperCase)  
 .sorted()  
 .forEach(System.out::println);Copy

This will give us the following output:

CARLA  
CARMEN  
CHRISTIANCopy

In some cases, this technique can be useful. Of course, we have to keep in mind that forEach() is a terminal operation, and so the stream will be consumed. Since a stream can only be consumed once, this can be an issue.

Moreover, if we add a null value to the list, then the output will become confusing again.

A better alternative consists of relying on the peek() method. This is an intermediate operation that executes a certain action on the current element and forwards the element to the next operation in the pipeline. The following diagram shows the peek() operation at work:

Diagrama

Descripción generada automáticamente

Let's see it in code form:

System.out.println("After:");  
  
names.stream()  
 .peek(p -> System.out.println("\tstream(): " + p))  
 .filter(s -> s.startsWith("c"))  
 .peek(p -> System.out.println("\tfilter(): " + p))  
 .map(String::toUpperCase)  
 .peek(p -> System.out.println("\tmap(): " + p))  
 .sorted()  
 .peek(p -> System.out.println("\tsorted(): " + p))  
 .collect(Collectors.toList());Copy

The following is an example of the output we may receive:

Tabla

Descripción generada automáticamente

Now, let's intentionally add a null value to the list and run it again:

List<String> names = Arrays.asList("anna", "bob",   
 "christian", null, "carmen", "rick", "carla");Copy

The following output was obtained after adding a null value to the list:

Texto, Carta

Descripción generada automáticamente

This time, we can see that a null value occurred after applying stream(). Since stream() is the first operation, we can easily figure out that the error resides in the list content.

180. Filtering the non-zero elements of a stream

In [Chapter 8](https://subscription.packtpub.com/book/programming/9781789801415/9/ch09lvl1sec36/39e6a5bc-3eaa-42ee-aff1-7e2e094a75d2.xhtml), *Functional Style Programming – Fundamentals and Design Patterns*, in the *Writing functional interfaces* section, we defined a filter() method based on a functional interface named Predicate. The Java Stream API already has such a method, and the functional interface is called java.util.function.Predicate.

Let's assume that we have the following List of integers:

List<Integer> ints = Arrays.asList(1, 2, -4, 0, 2, 0, -1, 14, 0, -1);Copy

Streaming this list and extracting only non-zero elements can be accomplished as follows:

List<Integer> result = ints.stream()  
 .filter(i -> i != 0)  
 .collect(Collectors.toList());Copy

The resulting list will contain the following elements: 1, 2, -4, 2, -1, 14, -1

The following diagram shows how filter() works internally:

Diagrama

Descripción generada automáticamente con confianza baja

Notice that, for several common operations, the Java Stream API already provides *out of the box* intermediate operations. Hence, there is no need to provide a Predicate. Some of these operations are as follows:

* distinct(): Removes duplicates from the stream
* skip(*n*): Discards the first *n* elements
* limit(*s*): Truncates the stream to be no longer than *s* in length
* sorted(): Sorts the stream according to the natural order
* sorted(Comparator<? super T> comparator): Sorts the stream according to the given Comparator

Let's add these operations and a filter() to an example. We will filter zeros, filter duplicates, skip 1 value, truncate the remaining stream to two elements, and sort them by their natural order:

List<Integer> result = ints.stream()  
 .filter(i -> i != 0)  
 .distinct()  
 .skip(1)  
 .limit(2)  
 .sorted()  
 .collect(Collectors.toList());Copy

The resulting list will contain the following two elements: **-4** and **2**.

The following diagram shows how this stream pipeline works internally:

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza media

When the filter() operation needs a complex/compound or long condition, then it is advisable to extract it in an ancillary static method and rely on *method references*. Therefore, avoid something like this:

List<Integer> result = ints.stream()  
 .filter(value -> value > 0 && value < 10 && value % 2 == 0)  
 .collect(Collectors.toList());Copy

You should prefer something like this (Numbers is the class containing the ancillary method):

List<Integer> result = ints.stream()  
 .filter(Numbers::evenBetween0And10)  
 .collect(Collectors.toList());  
  
private static boolean evenBetween0And10(int value) {  
 return value > 0 && value < 10 && value % 2 == 0;  
}Copy

181. Infinite streams, takeWhile(), and dropWhile()

In the first part of this problem, we will talk about infinite streams. In the second part, we will talk about the takeWhile() and dropWhile() APIs.

An infinite stream is a stream that creates data indefinitely. Because streams are lazy, they can be infinite. More precisely, creating an infinite stream is accomplished as an intermediate operation, and so no data is created until a terminal operation of the pipeline is executed.

For example, the following code will theoretically run forever. This behavior is triggered by the forEach() terminal operation and caused by a missing constraint or limitation:

Stream.iterate(1, i -> i + 1)  
 .forEach(System.out::println);Copy

The Java Stream API allows us to create and manipulate an infinite stream in several ways, as you will see shortly.

Moreover, a Stream can be *ordered* or *unordered*, depending on the defined *encounter order*. Whether or not a stream has an *encounter order* depends on the source of data and the intermediate operations. For example, a Stream that has a List as its source is ordered because List has an intrinsic ordering. On the other hand, a Stream that has a Set as its source is unordered because Set doesn't guarantee order. Some intermediate operations (for example, sorted()) may impose an order to an unordered Stream, while some terminal operations (for example, forEach()) may ignore the encounter order.

Commonly, the performance of sequential streams is insignificantly affected by ordering, but depending on the applied operations, the performance of parallel streams may be significantly affected by the presence of an ordered Stream.

Don't confuse Collection.stream().forEach() with Collection.forEach(). While Collection.forEach() can keep order by relying on the collection's iterator (if any), the Collection.stream().forEach() order undefined. For example, iterating a List several times via list.forEach() processes the elements in insertion order, while list.parallelStream().forEach() produces a different result at each run. As a rule of thumb, if a stream is not needed, then iterate over a collection via Collection.forEach().

We can turn an ordered stream into an unordered stream via BaseStream.unordered(), as shown in the following example:

List<Integer> list   
 = Arrays.asList(1, 4, 20, 15, 2, 17, 5, 22, 31, 16);  
  
Stream<Integer> unorderedStream = list.stream()  
 .unordered();Copy

Infinite sequential ordered stream

An infinite sequential ordered stream can be obtained via Stream.iterate​(T seed, UnaryOperator<T> f). The resulting stream starts from the specified seed and continues by applying the f function to the previous element (for example, the n element is f(n-1)).

For example, a stream of integers of type 1, 2, 3, ..., n can be created as follows:

Stream<Integer> infStream = Stream.iterate(1, i -> i + 1);Copy

Furthermore, we can use this stream for a variety of purposes. For example, let's use it to fetch a list of the first 10 even integers:

List<Integer> result = infStream  
 .filter(i -> i % 2 == 0)  
 .limit(10)  
 .collect(Collectors.toList());Copy

The List content will be as follows (notice that the infinite stream will create the elements 1, 2, 3, ..., 20, but only the following elements are matching our filter until the limit of 10 elements is reached):

2, 4, 6, 8, 10, 12, 14, 16, 18, 20Copy

Notice the presence of the limit() intermediate operation. Its presence is mandatory; otherwise, the code will run indefinitely. We must explicitly discard the stream; in other words, we must explicitly specify how many elements that match our filter should be collected in the final list. Once the limit has been reached, the infinite stream is discarded.

But let's assume that we don't want the list of the first 10 even integers, and we actually want the list of even integers until 10 (or any other limit). Starting with JDK 9, we can shape this behavior via a new flavor of Stream.iterate(). This flavor allows us to embed a hasNext predicate directly into the stream declaration (iterate​(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)). The stream terminates as soon as the hasNext predicate returns false:

Stream<Integer> infStream = Stream.iterate(  
 1, i -> i <= 10, i -> i + 1);Copy

This time, we can remove the limit() intermediate operation since our hasNext predicate imposes the limit of 10 elements:

List<Integer> result = infStream  
 .filter(i -> i % 2 == 0)  
 .collect(Collectors.toList());Copy

The resulting List is as follows (conforming to our hasNext predicate, the infinite stream creates the elements 1, 2, 3, ..., 10, but only the following five elements match our stream filter):

2, 4, 6, 8, 10Copy

Of course, we can combine this flavor of Stream.iterate() and limit() to shape more complex scenarios. For example, the following stream will create new element until the *have next* predicate, i -> i <= 10. Since we are using random values, the moment when the hasNext predicate will return false is nondeterministic:

Stream<Integer> infStream = Stream.iterate(  
 1, i -> i <= 10, i -> i + i % 2 == 0   
 ? new Random().nextInt(20) : -1 \* new Random().nextInt(10));Copy

One possible output for this stream is as follows:

1, -5, -4, -7, -4, -2, -8, -8, ..., 3, 0, 4, -7, -6, 10, ...Copy

Now, the following pipeline will collect a maximum of 25 numbers that were created via infStream:

List<Integer> result = infStream  
 .limit(25)  
 .collect(Collectors.toList());Copy

Now, the infinite stream can be discarded from two places. If the hasNext predicate returns false until we collect 25 elements, then we remain with the collected elements at that moment (less than 25). If the hasNext predicate doesn't return false until we collect 25 elements, then the limit() operation will discard the rest of the stream.

Unlimited stream of pseudorandom values

If we want to create unlimited streams of pseudorandom values, we can rely on the methods of Random, such as ints(), longs(), and doubles(). For example, an unlimited stream of pseudorandom integer values can be declared as follows (the generated integers will be in the [1, 100] range):

IntStream rndInfStream = new Random().ints(1, 100);Copy

Trying to fetch a list of 10 even pseudorandom integer values can rely on this stream:

List<Integer> result = rndInfStream  
 .filter(i -> i % 2 == 0)  
 .limit(10)  
 .boxed()  
 .collect(Collectors.toList());Copy

One possible output is as follows:

8, 24, 82, 42, 90, 18, 26, 96, 86, 86Copy

This time, it is harder to say how many numbers were actually generated until the aforementioned list is collected.

Another flavor of ints() is ints​(long streamSize, int randomNumberOrigin, int randomNumberBound). The first argument allows us to specify how many pseudorandom values should be generated. For example, the following stream will generate exactly 10 values in the range of [1, 100]:

IntStream rndInfStream = new Random().ints(10, 1, 100);Copy

We can fetch the even values from these 10, as follows:

List<Integer> result = rndInfStream  
 .filter(i -> i % 2 == 0)  
 .boxed()  
 .collect(Collectors.toList());Copy

One possible output is as follows:

80, 28, 60, 54Copy

We can use this example as a base for generating random strings of a fixed length, as follows:

IntStream rndInfStream = new Random().ints(20, 48, 126);  
String result = rndInfStream  
 .mapToObj(n -> String.valueOf((char) n))  
 .collect(Collectors.joining());Copy

One possible output is as follows:

AIW?F1obl3KPKMItqY8>Copy

Stream.ints() comes with two more flavors: one that doesn't take any argument (an unlimited stream of integers) and another that takes a single argument representing the number of values that should be generated, that is, ints​(long streamSize).

Infinite sequential unordered stream

In order to create an infinite sequential unordered stream, we can rely on Stream.generate​(Supplier<? extends T> s). In this case, each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, and so on.

For example, let's assume that we have a simple helper that generates passwords of eight characters:

private static String randomPassword() {  
  
 String chars = "abcd0123!@#$";  
  
 return new SecureRandom().ints(8, 0, chars.length())  
 .mapToObj(i -> String.valueOf(chars.charAt(i)))  
 .collect(Collectors.joining());  
}Copy

Furthermore, we want to define an infinite sequential unordered stream that returns random passwords (Main is the class that contains the preceding helper):

Supplier<String> passwordSupplier = Main::randomPassword;  
Stream<String> passwordStream = Stream.generate(passwordSupplier);Copy

At this point, passwordStream can create passwords indefinitely. But let's create 10 such passwords:

List<String> result = passwordStream  
 .limit(10)  
 .collect(Collectors.toList());Copy

One possible output is as follows:

213c1b1c, 2badc$21, d33321d$, @a0dc323, 3!1aa!dc, 0a3##@3!, $!b2#1d@, 0@0#dd$#, cb$12d2@, d2@@cc@dCopy

Take while a predicate returns true

One of the most useful methods that was added to the Stream class, starting with JDK 9, was takeWhile​(Predicate<? super T> predicate). This method comes with two different behaviors, as follows:

* If the stream is ordered, it returns a stream consisting of the *longest prefix of elements* taken from this stream that match the given predicate.
* If the stream is unordered, and some (but not all) of the elements of this stream match the given predicate, then the behavior of this operation is nondeterministic; it is free to take any subset of matching elements (which includes the empty set).

In the case of an ordered Stream, the *longest prefix of elements* is a contiguous sequence of elements of the stream that match with the given predicate.

Note that takeWhile() will discard the remaining stream once the given predicate returns false.

For example, fetching a list of 10 integers can be done as follows:

List<Integer> result = IntStream  
 .iterate(1, i -> i + 1)  
 .takeWhile(i -> i <= 10)  
 .boxed()  
 .collect(Collectors.toList());Copy

This will give us the following output:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10Copy

Alternatively, we could fetch a List of random even integers until the first generated value is less than 50:

List<Integer> result = new Random().ints(1, 100)  
 .filter(i -> i % 2 == 0)  
 .takeWhile(i -> i >= 50)  
 .boxed()  
 .collect(Collectors.toList());Copy

We can even join the predicates in takeWhile():

List<Integer> result = new Random().ints(1, 100)  
 .takeWhile(i -> i % 2 == 0 && i >= 50)  
 .boxed()  
 .collect(Collectors.toList());Copy

One possible output can be obtained as follows (it can be empty as well):

64, 76, 54, 68Copy

How about fetching a List of random passwords until the first generated password doesn't contain the ! character?

Well, based on the helper we listed earlier, we can do this like so:

List<String> result = Stream.generate(Main::randomPassword)  
 .takeWhile(s -> s.contains("!"))  
 .collect(Collectors.toList());Copy

One possible output can be obtained as follows (it can be empty as well):

0!dac!3c, 2!$!b2ac, 1d12ba1!Copy

Now, let's assume that we have an unordered stream of integers. The following snippet of code takes a subset of elements that are less than or equal to 10:

Set<Integer> setOfInts = new HashSet<>(  
 Arrays.asList(1, 4, 3, 52, 9, 40, 5, 2, 31, 8));  
  
List<Integer> result = setOfInts.stream()  
 .takeWhile(i -> i<= 10)  
 .collect(Collectors.toList());Copy

One possible output is as follows (remember that, for an unordered stream, the result is nondeterministic):

1, 3, 4Copy

Drop while a predicate returns true

Starting with JDK 9, we also have the Stream.dropWhile​(Predicate<? super T> predicate) method. This method is the opposite of takeWhile(). Instead of taking elements until the given predicate returns false, this method drops the elements until the given element returns false and includes the rest of the elements in the returned stream:

* If the stream is ordered, it returns a stream consisting of the remaining elements of this stream after dropping the *longest prefix of elements* that match the given predicate.
* If the stream is unordered, and some (but not all) of the elements of this stream match the given predicate, then the behavior of this operation is nondeterministic; it is free to drop any subset of matching elements (which includes the empty set).

In the case of an ordered Stream, the *longest prefix of elements* is a contiguous sequence of elements of the stream that match the given predicate.

For example, let's collect 5 integers after dropping the first 10:

List<Integer> result = IntStream  
 .iterate(1, i -> i + 1)  
 .dropWhile(i -> i <= 10)  
 .limit(5)  
 .boxed()  
 .collect(Collectors.toList());Copy

This will always give the following output:

11, 12, 13, 14, 15Copy

Alternatively, we can fetch a List of five random even integers greater than 50 (at least, this is what we may think the code does):

List<Integer> result = new Random().ints(1, 100)  
 .filter(i -> i % 2 == 0)  
 .dropWhile(i -> i < 50)  
 .limit(5)  
 .boxed()  
 .collect(Collectors.toList());Copy

One possible output is as follows:

78, 16, 4, 94, 26Copy

But why is 16 and 4 there? They are even, but not greater than 50! Well, they are there because they came after the first element, which failed the predicate. Mainly, we are dropping values while they are smaller than 50 (dropWhile(i -> i < 50)). The 78 value will fail this predicate, so dropWhile ends its job. Furthermore, all the generated elements are included in the result until limit(5) takes action.

Let's look at another similar trap. Let's fetch a List of five random passwords containing the ! character (at least, this is what we may think the code does):

List<String> result = Stream.generate(Main::randomPassword)  
 .dropWhile(s -> !s.contains("!"))  
 .limit(5)  
 .collect(Collectors.toList());Copy

One possible output is as follows:

bab2!3dd, c2@$1acc, $c1c@cb@, !b21$cdc, #b103c21Copy

Again, we can see passwords that don't contain the ! character. The bab2!3dd password will fail our predicate and will end up in the final result (List). The four generated passwords are added to the result without being influenced by dropWhile().

Now, let's assume that we have an unordered stream of integers. The following snippet of code drops a subset of elements that are less than or equal to 10 and keeps the rest:

Set<Integer> setOfInts = new HashSet<>(  
 Arrays.asList(5, 42, 3, 2, 11, 1, 6, 55, 9, 7));  
  
List<Integer> result = setOfInts.stream()  
 .dropWhile(i -> i <= 10)  
 .collect(Collectors.toList());Copy

One possible output is as follows (remember that, for an unordered stream, the result is nondeterministic):

55, 7, 9, 42, 11Copy

If all the elements match the given predicate, then takeWhile() takes and dropWhile() drops all elements (it doesn't matter if the stream is ordered or unordered). On the other hand, if none of the elements match the given predicate, then takeWhile() takes nothing (returns an empty stream) and dropWhile() drops nothing (returns the stream).

Avoid using take/dropWhile() in the context of parallel streams since they are expensive operations, especially for ordered streams. If it is suitable for the case, then just remove the ordering constraint via BaseStream.unordered().

182. Mapping the elements of a stream

Mapping the elements of a stream is an intermediate operation that's used for *transforming* these elements into a new version of them by applying the given function to each element and accumulating the results in a new Stream (for example, transforming a Stream<String> into a Stream<Integer>, or transforming a Stream<String> into another Stream<String>, and so on).

Using Stream.map()

Basically, we call Stream.map​(Function<? super T,​? extends R> mapper) to apply the mapper function on each element of the stream. The result is a new Stream. It doesn't modify the source Stream.

Let's assume that we have the following Melon class:

public class Melon {  
  
 private String type;  
 private int weight;  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}Copy

We also need to assume that we have List<Melon>:

List<Melon> melons = Arrays.asList(new Melon("Gac", 2000),  
 new Melon("Hemi", 1600), new Melon("Gac", 3000),  
 new Melon("Apollo", 2000), new Melon("Horned", 1700));Copy

Furthermore, we want to extract only the names of the melons in another list, List<String>.

For this task, we can rely on map(), as follows:

List<String> melonNames = melons.stream()  
 .map(Melon::getType)  
 .collect(Collectors.toList());Copy

The output will contain the following types of melons:

Gac, Hemi, Gac, Apollo, HornedCopy

The following diagram depicts how map() works for this example:

Diagrama

Descripción generada automáticamente

So, the map() method gets a Stream<Melon> and outputs a Stream<String>. Each Melon passes through the map() method, and this method extracts the melon's type (which is a String) and stores it in another Stream.

Similarly, we can extract the weights of melons. Since weights are integers, the map() method will return a Stream<Integer>:

List<Integer> melonWeights = melons.stream()  
 .map(Melon::getWeight)  
 .collect(Collectors.toList());Copy

The output will contain the following weights:

2000, 1600, 3000, 2000, 1700Copy

Beside map(), the Stream class also provides flavors for primitives such as mapToInt(), mapToLong(), and mapToDouble(). These methods return the int primitive specialization of Stream (IntStream), the long primitive specialization of Stream (LongStream) and the double primitive specialization of Stream (StreamDouble).

While map() can map the elements of a Stream to a new Stream via a Function, do not conclude that we can do the following:

List<Melon> lighterMelons = melons.stream()  
 .map(m -> m.setWeight(m.getWeight() - 500))  
 .collect(Collectors.toList());Copy

This will not work/compile because the setWeight() method returns void. In order to make it work, we need to return Melon, but this means we have to add some perfunctory code (for example, return):

List<Melon> lighterMelons = melons.stream()  
 .map(m -> {  
 m.setWeight(m.getWeight() - 500);  
  
 return m;  
 })  
 .collect(Collectors.toList());Copy

What do you think about the peek() temptation? Well, peek() stands for *look, but don't touch*, but it can be used to mutate state, as follows:

List<Melon> lighterMelons = melons.stream()  
 .peek(m -> m.setWeight(m.getWeight() - 500))  
 .collect(Collectors.toList());Copy

The output will contain the following melons (this looks good):

Gac(1500g), Hemi(1100g), Gac(2500g), Apollo(1500g), Horned(1200g)Copy

This is more clear than using map(). Calling setWeight() is a clear signal that we plan to mutate state, but the documentation specifies that the Consumer that's passed to peek() should be a *non-interfering* action (doesn't modify the data source of the stream).

For sequential streams (such as the preceding one), breaking this expectation can be kept under control without side effects; however, for parallel stream pipelines, the problem may become more complicated.

The action may be called at whatever time and in whatever thread the element is made available by the upstream operation, so if the action modifies the shared state, it is responsible for providing the required synchronization.

As a rule of thumb, think twice before using peek() to mutate the state. Also, be aware that this practice is a debate that falls under the bad practice or even anti-pattern umbrella.

Using Stream.flatMap()

As we just saw, map() knows how to wrap a sequence of elements in a Stream.

This means that map() can produce streams such as Stream<String[]>, Stream<List<String>>, Stream<Set<String>>, or even Stream<Stream<R>>.

But the problem is that these kinds of streams cannot be manipulated successfully (or, as we expected) by stream operations such as sum(), distinct(), filter(), and so on.

For example, let's consider the following array of Melon:

Melon[][] melonsArray = {  
 {new Melon("Gac", 2000), new Melon("Hemi", 1600)},   
 {new Melon("Gac", 2000), new Melon("Apollo", 2000)},   
 {new Melon("Horned", 1700), new Melon("Hemi", 1600)}  
};Copy

We can take this array and wrap it in a stream via Arrays.stream(), as shown in the following snippet of code:

Stream<Melon[]> streamOfMelonsArray = Arrays.stream(melonsArray);Copy

There are many other ways of obtaining a Stream of arrays. For example, if we have a string, s, then map(s -> s.split("")) will return a Stream<String[]>.

Now, we may think that obtaining the distinct Melon instances it is enough to call distinct(), as follows:

streamOfMelonsArray  
 .distinct()  
 .collect(Collectors.toList());Copy

But this is not going to work because distinct() will not look for a distinct Melon; instead, it will look for a distinct array Melon[] because this is what we have in the stream.

Moreover, the result that was returned in this case is of the Stream<Melon[]> type, not of the Stream<Melon> type. The final result will collect Stream<Melon[]> in List<Melon[]>.

How we can fix this problem?

We may consider applying Arrays.stream() in order to convert the Melon[] into a Stream<Melon>:

streamOfMelonsArray  
 .map(Arrays::stream) // Stream<Stream<Melon>>  
 .distinct()  
 .collect(Collectors.toList());Copy

Again, map() will not do what we might think it will do.

First, calling Arrays.stream() will return a Stream<Melon> from each of the given Melon[]. However, map() returns a Stream of elements, and so it will wrap the results of applying Arrays.stream() into a Stream. It will end up in a Stream<Stream<Melon>>.

So, this time, distinct() tries to detect distinct Stream<Melon> elements:

Diagrama

Descripción generada automáticamente

In order to fix this problem, we must rely on flatMap(). The following diagram depicts how flatMap() works internally:

Diagrama

Descripción generada automáticamente

Unlike map(), this method returns a stream by flattening all the separated streams. So, all the arrays will end up in the same stream:

streamOfMelonsArray  
 .flatMap(Arrays::stream) // Stream<Melon>  
 .distinct()  
 .collect(Collectors.toList());Copy

The output will contain distinct melons according to the Melon.equals() implementation:

Gac(2000g), Hemi(1600g), Apollo(2000g), Horned(1700g)Copy

Now, let's try another problem, starting with a List<List<String>>, as follows:

List<List<String>> melonLists = Arrays.asList(  
 Arrays.asList("Gac", "Cantaloupe"),  
 Arrays.asList("Hemi", "Gac", "Apollo"),  
 Arrays.asList("Gac", "Hemi", "Cantaloupe"),  
 Arrays.asList("Apollo"),  
 Arrays.asList("Horned", "Hemi"),  
 Arrays.asList("Hemi"));Copy

We try to obtain the distinct names of melons from this list. If wrapping an array into a stream can be done via Arrays.stream(), for a collection, we have Collection.stream(). Therefore, the first attempt may look as follows:

melonLists.stream()  
 .map(Collection::stream)  
 .distinct();Copy

But based on the previous problem, we already know that this will not work because map() will return Stream<Stream<String>>.

The solution is provided by flatMap(), as follows:

List<String> distinctNames = melonLists.stream()  
 .flatMap(Collection::stream)  
 .distinct()  
 .collect(Collectors.toList());Copy

The output is as follows:

Gac, Cantaloupe, Hemi, Apollo, HornedCopy

Beside flatMap(), the Stream class also provides flavors for primitives such as flatMapToInt(), flatMapToLong(), and flatMapToDouble(). These methods return the int primitive specialization of Stream (IntStream), the long primitive specialization of Stream (LongStream), and the double primitive specialization of Stream (StreamDouble).

183. Finding elements in a stream

Besides using filter(), which allows us to filter elements of a stream by a predicate, we can find an element in a stream via anyFirst() and findFirst().

Let's assume that we have the following list wrapped in a stream:

List<String> melons = Arrays.asList(  
 "Gac", "Cantaloupe", "Hemi", "Gac", "Gac",   
 "Hemi", "Cantaloupe", "Horned", "Hemi", "Hemi");Copy

findAny

The findAny() method returns an arbitrary (nondeterministic) element from the stream. For example, the following snippet of code will return an element from the preceding list:

Optional<String> anyMelon = melons.stream()  
 .findAny();  
  
if (!anyMelon.isEmpty()) {  
 System.out.println("Any melon: " + anyMelon.get());  
} else {  
 System.out.println("No melon was found");  
}Copy

Notice that there is no guarantee that it will return the same element at each execution. This statement is true especially in the case of parallelizing the stream.

We can combine findAny() with other operations as well. Here's an example:

String anyApollo = melons.stream()  
 .filter(m -> m.equals("Apollo"))  
 .findAny()  
 .orElse("nope");Copy

This time, the result will be nope. There is no Apollo in the list, and so the filter() operation will produce an empty stream. Furthermore, findAny() will return an empty stream as well, so orElse() will return the final result as the specified string, nope.

findFirst

If findAny() returns any element, findFirst() returns the first element from the stream. Obviously, this method is useful when we are interested only in the first element of a stream (for example, the winner of a contest should be the first element in a sorted list of competitors).

Nevertheless, if the stream has no *encounter order*, then any element may be returned. According to the documentation, *streams may or may not have a defined encounter order. It depends on the source and intermediate operations*. The same rule applies in parallelism as well.

For now, let's assume that we want the first melon in the list:

Optional<String> firstMelon = melons.stream()  
 .findFirst();  
  
if (!firstMelon.isEmpty()) {  
 System.out.println("First melon: " + firstMelon.get());  
} else {  
 System.out.println("No melon was found");  
}Copy

The output will be as follows:

First melon: GacCopy

We can combine findFirst() with other operations as well. Here's an example:

String firstApollo = melons.stream()  
 .filter(m -> m.equals("Apollo"))  
 .findFirst()  
 .orElse("nope");Copy

This time, the result will be nope since the filter() will produce an empty stream.

The following is another problem with integers (just follow the right-hand comments to quickly discover the flow):

List<Integer> ints = Arrays.asList(4, 8, 4, 5, 5, 7);  
  
int result = ints.stream()  
 .map(x -> x \* x - 1) // 23, 63, 23, 24, 24, 48  
 .filter(x -> x % 2 == 0) // 24, 24, 48  
 .findFirst() // 24  
 .orElse(-1);Copy

184. Matching elements in a stream

To match certain elements in a Stream, we can rely on the following methods:

* anyMatch()
* noneMatch()
* allMatch()

All of these methods take a Predicate as an argument and fetch a boolean result against it.

These three operations rely on the *short-circuiting* technique. In other words, these methods may return until we process the entire stream. For example, if allMatch() matches false (evaluates the given Predicate as false), then there is no reason to continue. The final result is false.

Let's assume that we have the following list wrapped in a stream:

List<String> melons = Arrays.asList(  
 "Gac", "Cantaloupe", "Hemi", "Gac", "Gac", "Hemi",   
 "Cantaloupe", "Horned", "Hemi", "Hemi");Copy

Now, let's try to answer the following questions:

* Does an element match the Gac string? Let's see that in the following code:

boolean isAnyGac = melons.stream()  
 .anyMatch(m -> m.equals("Gac")); // trueCopy

* Does an element match the Apollo string? Let's see that in the following code:

boolean isAnyApollo = melons.stream()  
 .anyMatch(m -> m.equals("Apollo")); // falseCopy

As a general question – is there an element in the stream that matches the given predicate?

* Do no elements match the Gac string? Let's see that in the following code:

boolean isNoneGac = melons.stream()  
 .noneMatch(m -> m.equals("Gac")); // falseCopy

* Do no elements match the Apollo string? Let's see that in the following code:

boolean isNoneApollo = melons.stream()  
 .noneMatch(m -> m.equals("Apollo")); // trueCopy

As a general question – are there no elements in the stream that match the given predicate?

* Do all the elements match the Gac string? Let's see that in the following code:

boolean areAllGac = melons.stream()  
 .allMatch(m -> m.equals("Gac")); // falseCopy

* Are all the elements larger than 2? Let's see that in the following code:

boolean areAllLargerThan2 = melons.stream()  
 .allMatch(m -> m.length() > 2);Copy

As a general question—do all the elements in the stream match the given predicate?

185. Sum, max, and min in a stream

Let's assume that we have the following Melon class:

public class Melon {  
  
 private String type;  
 private int weight;  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}Copy

Let's also assume that we have the following list of Melon wrapped in a stream:

List<Melon> melons = Arrays.asList(new Melon("Gac", 2000),  
 new Melon("Hemi", 1600), new Melon("Gac", 3000),  
 new Melon("Apollo", 2000), new Melon("Horned", 1700));Copy

Let's work on the Melon class using the sum(), min(), and max() terminal operations.

The sum(), min(), and max() terminal operations

Now, let's combine the elements of this stream to express the following queries:

* How can we calculate the total weight of melons (sum())?
* What is the heaviest melon (max())?
* What is the lightest melon (min())?

In order to calculate the total weight of melons, we need to sum up all the weights. For primitive specializations of Stream (IntStream, LongStream, and so on), the Java Stream API exposes a terminal operation named sum(). As its name suggests, this method sums up the elements of the stream:

int total = melons.stream()  
 .mapToInt(Melon::getWeight)  
 .sum();Copy

After sum(), we also have the max() and min() terminal operations. Obviously, max() returns the maximum value of the stream, while min() is its opposite:

int max = melons.stream()  
 .mapToInt(Melon::getWeight)  
 .max()  
 .orElse(-1);  
  
int min = melons.stream()  
 .mapToInt(Melon::getWeight)  
 .min()  
 .orElse(-1);Copy

The max() and min() operations return an OptionalInt (such as OptionalLong). If the maximum or minimum cannot be calculated (for example, in the case of an empty stream) then we choose to return -1. Since we are working with weights, and with positive numbers by their nature, returning -1 makes sense. But don't take this as a rule. Depending on the case, another value should be returned, or maybe using orElseGet()/orElseThrow() would be better.

For non-primitive specializations, check out the *Summarization collectors* section of this chapter.

Let's learn about reducing in the next section.

Reducing

sum(), max(), and min() are known as special cases of *reduction*. By *reduction*, we mean an abstraction based on two main statements:

* Take an initial value (T)
* Take a BinaryOperator<T> to combine two elements and produce a new value

Reductions can be accomplished via a terminal operation named reduce(), which follows this abstraction and defines two signatures (the second one doesn't use an initial value):

* T reduce​(T identity, BinaryOperator<T> accumulator)
* Optional<T> reduce​(BinaryOperator<T> accumulator)

With that being said, we can rely on the reduce() terminal operation to compute the sum of the elements, as follows (the initial value is 0, and the lambda is (m1, m2) -> m1 + m2)):

int total = melons.stream()  
 .map(Melon::getWeight)  
 .reduce(0, (m1, m2) -> m1 + m2);Copy

The following diagram depicts how the reduce() operation works:

Diagrama

Descripción generada automáticamente

So, how does the reduce() operation work?

Let's take a look at the following steps to figure this out:

1. First, 0 is used as the first parameter of the lambda (m1), and 2,000 is consumed from the stream and used as the second parameter (m2). 0 + 2000 produces 2000, and this becomes the new accumulated value.
2. Then, the lambda is called again with the accumulated value and the next element of the stream, 1,600, which produces the new accumulated value, 3,600.
3. Moving forward, the lambda is called again with the accumulated value and the next element, 3,000, which produces 6,600.
4. If we step forward again, the lambda is called again with the accumulated value and the next element, 2,000, which produces 8,600.
5. Finally, the lambda is called with 8,600 and the last element of the stream, 1,700, which produces the final value, 10,300.

The maximum and minimum can be calculated as well:

int max = melons.stream()  
 .map(Melon::getWeight)  
 .reduce(Integer::max)  
 .orElse(-1);  
  
int min = melons.stream()  
 .map(Melon::getWeight)  
 .reduce(Integer::min)  
 .orElse(-1);Copy

The advantage of using reduce() is that we can easily change the computation by simply passing another lambda. For example, we can quickly replace the sum with the product, as shown in the following example:

List<Double> numbers = Arrays.asList(1.0d, 5.0d, 8.0d, 10.0d);  
  
double total = numbers.stream()  
 .reduce(1.0 d, (x1, x2) -> x1 \* x2);Copy

Nevertheless, pay attention to cases that can lead to unwanted results. For example, if we want to compute the harmonic mean of the given numbers then there is not an *out of the box* special case of *reduction*, and so we can only rely on reduce(), as follows:

List<Double> numbers = Arrays.asList(1.0d, 5.0d, 8.0d, 10.0d);Copy

The harmonic mean formula is as follows:

Diagrama, Esquemático

Descripción generada automáticamente

In our case, *n* is the size of the list and *H* is 2.80701. Using a naive reduce() function will look as follows:

double hm = numbers.size() / numbers.stream()  
 .reduce((x1, x2) -> (1.0d / x1 + 1.0d / x2))  
 .orElseThrow();Copy

This will produce 3.49809.

This explanation relies on how we have expressed the calculation. In the first step, we calculate 1.0/1.0 + 1.0/5.0 = 1.2. Then, we may expect to do 1.2 + 1.0/1.8, but actually, the calculation is 1.0/1.2 + 1.0/1.8. Obviously, this is not what we want.

We can fix this by using mapToDouble(), as follows:

double hm = numbers.size() / numbers.stream()  
 .mapToDouble(x -> 1.0d / x)  
 .reduce((x1, x2) -> (x1 + x2))  
 .orElseThrow();Copy

This will produce the expected result, that is, 2.80701.

186. Collecting the result of a stream

Let's assume that we have the following Melon class:

public class Melon {  
  
 private String type;  
 private int weight;  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}Copy

Let's also assume that we have the List of Melon:

List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 2000),  
 new Melon("Hemi", 1600), new Melon("Gac", 3000),  
 new Melon("Apollo", 2000), new Melon("Horned", 1700),  
 new Melon("Gac", 3000), new Melon("Cantaloupe", 2600));Copy

Typically, a stream pipeline ends up with a summary of the elements in the stream. In other words, we need to collect the results in a data structure such as List, Set, or Map (and their companions).

For accomplishing this task, we can rely on the Stream.collect​(Collector<? super T,​A,​R> collector) method. This method gets a single argument representing a java.util.stream.Collector or a user-defined Collector.

The most famous collectors include the following:

* toList()
* toSet()
* toMap()
* toCollection()

Their names speak for themselves. Let's take a look at several examples:

* Filter melons that are heavier than 1,000 g and collect the result in a List via toList() and toCollection():

List<Integer> resultToList = melons.stream()  
 .map(Melon::getWeight)  
 .filter(x -> x >= 1000)  
 .collect(Collectors.toList());  
  
List<Integer> resultToList = melons.stream()  
 .map(Melon::getWeight)  
 .filter(x -> x >= 1000)  
 .collect(Collectors.toCollection(ArrayList::new));Copy

The argument of the toCollection() method is a Supplier that provides a new empty Collection into which the results will be inserted.

* Filter melons that are heavier than 1,000 g and collect the result without duplicates in a Set via toSet() and toCollection():

Set<Integer> resultToSet = melons.stream()  
 .map(Melon::getWeight)  
 .filter(x -> x >= 1000)  
 .collect(Collectors.toSet());  
  
Set<Integer> resultToSet = melons.stream()  
 .map(Melon::getWeight)  
 .filter(x -> x >= 1000)  
 .collect(Collectors.toCollection(HashSet::new));Copy

* Filter melons that are heavier than 1,000 grams, collect the result without duplicates, and sort into ascending order in a Set via toCollection():

Set<Integer> resultToSet = melons.stream()  
 .map(Melon::getWeight)  
 .filter(x -> x >= 1000)  
 .collect(Collectors.toCollection(TreeSet::new));Copy

* Filter a distinct Melon and collect the result in a Map<String, Integer> via toMap():

Map<String, Integer> resultToMap = melons.stream()  
 .distinct()  
 .collect(Collectors.toMap(Melon::getType,   
 Melon::getWeight));Copy

The two arguments of the toMap() method represent a mapping function that's used to produce keys and their respective values (this is prone to the java.lang.IllegalStateException duplicate key exception if two Melon have the same key).

* Filter a distinct Melon and collect the result in a Map<Integer, Integer> via toMap() using random keys (prone to the java.lang.IllegalStateException duplicate key if two identical keys are generated):

Map<Integer, Integer> resultToMap = melons.stream()  
 .distinct()  
 .map(x -> Map.entry(  
 new Random().nextInt(Integer.MAX\_VALUE), x.getWeight()))  
 .collect(Collectors.toMap(Entry::getKey, Entry::getValue));Copy

* Collect a Melon in a map via toMap() and avoid the potential java.lang.IllegalStateException duplicate key by choosing the existing (old) value in case of a key collision:

Map<String, Integer> resultToMap = melons.stream()  
 .collect(Collectors.toMap(Melon::getType, Melon::getWeight,  
 (oldValue, newValue) -> oldValue));Copy

The last argument of the toMap() method is a merge function and is used to resolve collisions between values associated with the same key, as supplied to Map.merge(Object, Object, BiFunction).

Obviously, choosing the new value can be done with (oldValue, newValue) -> newValue:

* Put the preceding example into a sorted Map (for example, by weight):

Map<String, Integer> resultToMap = melons.stream()  
 .sorted(Comparator.comparingInt(Melon::getWeight))  
 .collect(Collectors.toMap(Melon::getType, Melon::getWeight,  
 (oldValue, newValue) -> oldValue,  
 LinkedHashMap::new));Copy

The last argument of this toMap() flavor represents a Supplier that provides a new empty Map into which the results will be inserted. In this example, this Supplier is needed to preserve the order after sorting. Since HashMap doesn't guarantee the order of insertion, we need to rely on LinkedHashMap.

* Collect the word frequency count via toMap():

String str = "Lorem Ipsum is simply   
 Ipsum Lorem not simply Ipsum";  
  
Map<String, Integer> mapOfWords = Stream.of(str)  
 .map(w -> w.split("\\s+"))  
 .flatMap(Arrays::stream)  
 .collect(Collectors.toMap(  
 w -> w.toLowerCase(), w -> 1, Integer::sum));Copy

Beside toList(), toMap(), and toSet(), the Collectors class also exposes collectors to unmodifiable and concurrent collections such as toUnmodifiableList(), toConcurrentMap(), and so on.

187. Joining the results of a stream

Let's assume that we have the following Melon class:

public class Melon {  
  
 private String type;  
 private int weight;  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}Copy

Let's also assume that we have the List of Melon:

List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 2000),  
 new Melon("Hemi", 1600), new Melon("Gac", 3000),  
 new Melon("Apollo", 2000), new Melon("Horned", 1700),  
 new Melon("Gac", 3000), new Melon("Cantaloupe", 2600));Copy

In the previous problem, we talked about the Stream API that's built into Collectors. In this category, we also have Collectors.joining(). The goal of these collectors is to concatenate the elements of a stream into a String in the *encounter order*. Optionally, these collectors can use a delimiter, a prefix, and a suffix, and so the most comprehensive joining() flavor is String joining​(CharSequence delimiter, CharSequence prefix, CharSequence suffix).

But if all we want is to concatenate the names of melons without a delimiter, then this is the way to go (just for fun, let's sort and remove the duplicates as well):

String melonNames = melons.stream()  
 .map(Melon::getType)  
 .distinct()  
 .sorted()  
 .collect(Collectors.joining());Copy

We will receive the following output:

ApolloCantaloupeCrenshawGacHemiHornedCopy

A nicer solution consists of adding a delimiter, for example, a comma and a space:

String melonNames = melons.stream()  
 ...  
 .collect(Collectors.joining(", "));Copy

We will receive the following output:

Apollo, Cantaloupe, Crenshaw, Gac, Hemi, HornedCopy

We can also enrich the output with a prefix and suffix:

String melonNames = melons.stream()  
 ...  
 .collect(Collectors.joining(", ",   
 "Available melons: ", " Thank you!"));Copy

We will receive the following output:

Available melons: Apollo, Cantaloupe, Crenshaw, Gac, Hemi, Horned Thank you!Copy

188. Summarization collectors

Let's assume that we have the well-known Melon class (that uses type and weight) and List of Melon:

List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 2000),  
 new Melon("Hemi", 1600), new Melon("Gac", 3000),  
 new Melon("Apollo", 2000), new Melon("Horned", 1700),  
 new Melon("Gac", 3000), new Melon("Cantaloupe", 2600));Copy

The Java Stream API groups the count, sum, min, average, and max operations under the term *summarization*. The methods dedicated to performing *summarization* operations are found in the Collectors class.

We'll take a look at all of these operations in the following sections.

Summing

Let's assume that we want to sum all the weights of melons. We did this in the *Sum, max, and min in a stream* section via primitive specializations of Stream. Now, let's do it via the summingInt​(ToIntFunction<? super T> mapper) collector:

int sumWeightsGrams = melons.stream()  
 .collect(Collectors.summingInt(Melon::getWeight));Copy

So, Collectors.summingInt() is a factory method that takes a function that's capable of mapping an object into an int that has to be summed as an argument. A collector is returned that performs the *summarization* via the collect() method. The following diagram depicts how summingInt() works:

Diagrama

Descripción generada automáticamente

While traversing the stream, each weight (Melon::getWeight) is mapped to its number, and this number is added to an accumulator, starting from the initial value, that is, 0.

After summingInt(), we have summingLong() and summingDouble(). How do we sum up the weights of melons in kilograms? This can be accomplished via summingDouble(), as follows:

double sumWeightsKg = melons.stream()  
 .collect(Collectors.summingDouble(  
 m -> (double) m.getWeight() / 1000.0d));Copy

If we just need the result in kilograms, we can still perform the sum in grams, as follows:

double sumWeightsKg = melons.stream()  
 .collect(Collectors.summingInt(Melon::getWeight)) / 1000.0d;Copy

Since *summarizations* are actually *reductions*, the Collectors class provides a reducing() method as well. Obviously, this method has a more general utilization, allowing us to provide all kinds of lambdas via its three flavors:

* reducing​(BinaryOperator<T> op)
* reducing​(T identity, BinaryOperator<T> op)
* reducing​(U identity, Function<? super T,​? extends U> mapper, BinaryOperator<U> op)

The arguments of reducing() are pretty straightforward. We have the identity value for the reduction (as well as the value that is returned when there are no input elements), a mapping function to apply to each input value, and a function that's used to reduce the mapped values.

For example, let's rewrite the preceding snippet of code via reducing(). Notice that we start the sum from 0, convert it from grams into kilograms via a mapping function, and reduce the values (the resulted kilograms) via a lambda:

double sumWeightsKg = melons.stream()  
 .collect(Collectors.reducing(0.0,  
 m -> (double) m.getWeight() / 1000.0d, (m1, m2) -> m1 + m2));Copy

Alternatively, we can simply convert to kilograms at the end:

double sumWeightsKg = melons.stream()  
 .collect(Collectors.reducing(0,  
 m -> m.getWeight(), (m1, m2) -> m1 + m2)) / 1000.0d;Copy

Rely on reducing() whenever there is no suitable built-in solution. Think of reducing() as a *generalized summarization*.

Averaging

How about computing the average weight of a melon?

For this, we have Collectors.averagingInt(), averagingLong(), and averagingDouble():

double avgWeights = melons.stream()  
 .collect(Collectors.averagingInt(Melon::getWeight));Copy

Counting

Counting the number of words in a piece of text is a common problem that can be solved by count():

String str = "Lorem Ipsum is simply dummy text ...";  
  
long numberOfWords = Stream.of(str)  
 .map(w -> w.split("\\s+"))  
 .flatMap(Arrays::stream)  
 .filter(w -> w.trim().length() != 0)  
 .count();Copy

But let's see how many Melon weighing 3,000 there are in our stream:

long nrOfMelon = melons.stream()  
 .filter(m -> m.getWeight() == 3000)  
 .count();Copy

We can use the collector that's returned by the counting() factory method:

long nrOfMelon = melons.stream()  
 .filter(m -> m.getWeight() == 3000)  
 .collect(Collectors.counting());Copy

We can also use the clumsy approach of using reducing():

long nrOfMelon = melons.stream()  
 .filter(m -> m.getWeight() == 3000)  
 .collect(Collectors.reducing(0L, m -> 1L, Long::sum));Copy

Maximum and minimum

In the *Sum, max, and min in a stream* section, we already computed the minimum and maximum value via the min() and max() methods. This time, let's compute the heaviest and the lightest Melon via the Collectors.maxBy() and Collectors.minBy() collectors. These collectors take a Comparator as an argument to compare the elements in the stream and return an Optional (this Optional will be empty if the stream is empty):

Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);  
  
Melon heaviestMelon = melons.stream()  
 .collect(Collectors.maxBy(byWeight))  
 .orElseThrow();  
  
Melon lightestMelon = melons.stream()  
 .collect(Collectors.minBy(byWeight))  
 .orElseThrow();Copy

In this case, if the stream is empty, we just throw NoSuchElementException.

Getting all

Is there a way to obtain the count, sum, average, min, and max in a single unitary operation?

Yes, there is! Whenever we need two or more of these operations, we can rely on Collectors.summarizingInt​(), summarizingLong(), and summarizingDouble(). These methods wrap these operations in IntSummaryStatistics, LongSummaryStatistics, and DoubleSummaryStatistics, respectively, as follows:

IntSummaryStatistics melonWeightsStatistics = melons  
 .stream().collect(Collectors.summarizingInt(Melon::getWeight));Copy

Printing this object produces the following output:

IntSummaryStatistics{count=7, sum=15900, min=1600, average=2271.428571, max=3000}Copy

For each of these operations, we have dedicated getters:

int max = melonWeightsStatistics.getMax()Copy

We're all done! Now, let's talk about grouping elements of a stream.

189. Grouping

Let's assume that we have the following Melon class and List of Melon:

public class Melon {  
  
 enum Sugar {  
 LOW, MEDIUM, HIGH, UNKNOWN  
 }  
  
 private final String type;  
 private final int weight;  
 private final Sugar sugar;  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}  
  
List<Melon> melons = Arrays.asList(  
 new Melon("Crenshaw", 1200),  
 new Melon("Gac", 3000), new Melon("Hemi", 2600),  
 new Melon("Hemi", 1600), new Melon("Gac", 1200),  
 new Melon("Apollo", 2600), new Melon("Horned", 1700),  
 new Melon("Gac", 3000), new Melon("Hemi", 2600)  
);Copy

The Java Stream API exposes the same functionality as the SQL GROUP BY clause via Collectors.groupingBy().

While the SQL GROUP BY clause works on database tables, Collectors.groupingBy() works on elements of streams.

In other words, the groupingBy() methods are capable of grouping elements with certain distinguishing characteristics. Before streams and functional-style programming (Java 8), such tasks were applied to collections via a bunch of *spaghetti* code that was cumbersome, verbose, and error-prone. Starting with Java 8, we have *grouping collectors*.

Let's take a look at single-level grouping and multilevel grouping in the next section. We will start with single-level grouping.

Single-level grouping

All grouping collectors have a *classification function* (the function that classifies the elements of the stream into different groups). Mainly, this is an instance of the Function<T, R> functional interface.

Each element of the stream (of the T type) is passed through this function, and the return will be a *classifier object* (of the R type). All the returned R types represent the keys (K) of a Map<K, V>, and each group is a value in this Map<K, V>.

In other words, the key (K) is the value returned by the classification function, and the value (V) is a list of elements in the stream that have this classified value (K). So, the final result is of the Map<K, List<T>> type.

Let's look at an example to bring some light to this brain-teasing explanation. This example relies on the simplest flavor of groupingBy(), that is, groupingBy​(Function<? super T,​? extends K> classifier).

So, let's group Melon by type:

Map<String, List<Melon>> byTypeInList = melons.stream()  
 .collect(groupingBy(Melon::getType));Copy

The output will be as follows:

{  
 Crenshaw = [Crenshaw(1200 g)],  
 Apollo = [Apollo(2600 g)],  
 Gac = [Gac(3000 g), Gac(1200 g), Gac(3000 g)],  
 Hemi = [Hemi(2600 g), Hemi(1600 g), Hemi(2600 g)],  
 Horned = [Horned(1700 g)]  
}Copy

We can also group Melon by weight:

Map<Integer, List<Melon>> byWeightInList = melons.stream()  
 .collect(groupingBy(Melon::getWeight));Copy

The output will be as follows:

{  
 1600 = [Hemi(1600 g)],  
 1200 = [Crenshaw(1200 g), Gac(1200 g)],  
 1700 = [Horned(1700 g)],  
 2600 = [Hemi(2600 g), Apollo(2600 g), Hemi(2600 g)],  
 3000 = [Gac(3000 g), Gac(3000 g)]  
}Copy

This grouping is shown in the following diagram. More precisely, this is a snapshot of the moment when Gac(1200 g) passes through the classification function (Melon::getWeight):

Diagrama

Descripción generada automáticamente

So, in the melon-classification example, a key is the weight of Melon, and its value is a list containing all the Melon objects of that weight.

The classification function can be a method reference or any other lambda.

One issue with the preceding approach is the presence of unwanted duplicates. This happens because the values are collected in a List (for example, 3000=[Gac(3000g), Gac(3000g)). But we can fix this by relying on another flavor of groupingBy(), that is, groupingBy​(Function<? super T,​? extends K> classifier, Collector<? super T,​A,​D> downstream).

This time, we can specify the desired downstream collector as the second argument. So, besides the classification function, we have a downstream collector as well.

If we wish to reject duplicates, we can use Collectors.toSet(), as follows:

Map<String, Set<Melon>> byTypeInSet = melons.stream()  
 .collect(groupingBy(Melon::getType, toSet()));Copy

The output is as follows:

{  
 Crenshaw = [Crenshaw(1200 g)],  
 Apollo = [Apollo(2600 g)],  
 Gac = [Gac(1200 g), Gac(3000 g)],  
 Hemi = [Hemi(2600 g), Hemi(1600 g)],  
 Horned = [Horned(1700 g)]  
}Copy

We can also do this by weight:

Map<Integer, Set<Melon>> byWeightInSet = melons.stream()  
 .collect(groupingBy(Melon::getWeight, toSet()));Copy

The output will be as follows:

{  
 1600 = [Hemi(1600 g)],  
 1200 = [Gac(1200 g), Crenshaw(1200 g)],  
 1700 = [Horned(1700 g)],  
 2600 = [Hemi(2600 g), Apollo(2600 g)],  
 3000 = [Gac(3000 g)]  
}Copy

Of course, in this case, distinct() can be used as well:

Map<String, List<Melon>> byTypeInList = melons.stream()  
 .distinct()  
 .collect(groupingBy(Melon::getType));Copy

The same goes for doing this by weight:

Map<Integer, List<Melon>> byWeightInList = melons.stream()  
 .distinct()  
 .collect(groupingBy(Melon::getWeight));Copy

Well, there are no more duplicates, but the results are not ordered. It would be nice to have this map ordered by keys, so the default HashMap is not very useful. If we could specify a TreeMap instead of the default HashMap, then the problem will be solved. We can do this via another flavor of groupingBy(), that is, groupingBy​(Function<? super T,​? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,​A,​D> downstream).

The second argument of this flavor allows us to provide a Supplier object that provides a new empty Map into which the results will be inserted:

Map<Integer, Set<Melon>> byWeightInSetOrdered = melons.stream()  
 .collect(groupingBy(Melon::getWeight, TreeMap::new, toSet()));Copy

Now, the output is ordered:

{  
 1200 = [Gac(1200 g), Crenshaw(1200 g)],  
 1600 = [Hemi(1600 g)],  
 1700 = [Horned(1700 g)],  
 2600 = [Hemi(2600 g), Apollo(2600 g)],  
 3000 = [Gac(3000 g)]  
}Copy

We can also have a List<Integer> containing the weights of 100 melons:

List<Integer> allWeights = new ArrayList<>(100);Copy

We want to split this list into 10 lists of 10 weights each. Basically, we can obtain this via grouping, as follows (we can apply parallelStream() as well):

final AtomicInteger count = new AtomicInteger();  
Collection<List<Integer>> chunkWeights = allWeights.stream()  
 .collect(Collectors.groupingBy(c -> count.getAndIncrement() / 10))  
 .values();Copy

Now, let's tackle another issue. By default, Stream<Melon> is divided into a suite of List<Melon>. But what can we do to divide Stream<Melon> into a suite of List<String>, where each list is holding only the types of melons, not the Melon instances?

Well, transforming elements of a stream is commonly the job of map(). But inside groupingBy(), this is the job of Collectors.mapping() (more details can be found in the *Filtering, flattening, and mapping collectors* section of this chapter):

Map<Integer, Set<String>> byWeightInSetOrdered = melons.stream()  
 .collect(groupingBy(Melon::getWeight, TreeMap::new,  
 mapping(Melon::getType, toSet())));Copy

This time, the output is exactly what we wanted:

{  
 1200 = [Crenshaw, Gac],  
 1600 = [Hemi],  
 1700 = [Horned],  
 2600 = [Apollo, Hemi],  
 3000 = [Gac]  
}Copy

Ok, so far, so good! Now, let's focus on the fact that two of the three flavors of groupingBy() accept a collector as an argument (for example, toSet()). This can be any collector. For example, we may want to group melons by types and count them. For this, Collectors.counting() is very helpful (more details can be found in the *Summarization collectors* section):

Map<String, Long> typesCount = melons.stream()  
 .collect(groupingBy(Melon::getType, counting()));Copy

The output will be as follows:

{Crenshaw=1, Apollo=1, Gac=3, Hemi=3, Horned=1}Copy

We can also do this by weight:

Map<Integer, Long> weightsCount = melons.stream()  
 .collect(groupingBy(Melon::getWeight, counting()));Copy

The output will be as follows:

{1600=1, 1200=2, 1700=1, 2600=3, 3000=2}Copy

Can we group the lightest and heaviest melons by type? Of course we can! We can do this via Collectors.minBy() and maxBy(), which were presented in the *Summarization collectors* section:

Map<String, Optional<Melon>> minMelonByType = melons.stream()  
 .collect(groupingBy(Melon::getType,  
 minBy(comparingInt(Melon::getWeight))));Copy

The output will be as follows (notice that minBy() returns an Optional):

{  
 Crenshaw = Optional[Crenshaw(1200 g)],  
 Apollo = Optional[Apollo(2600 g)],  
 Gac = Optional[Gac(1200 g)],  
 Hemi = Optional[Hemi(1600 g)],  
 Horned = Optional[Horned(1700 g)]  
}Copy

We can also do this via maxMelonByType():

Map<String, Optional<Melon>> maxMelonByType = melons.stream()  
 .collect(groupingBy(Melon::getType,  
 maxBy(comparingInt(Melon::getWeight))));Copy

The output will be as follows (notice that maxBy() returns an Optional):

{  
 Crenshaw = Optional[Crenshaw(1200 g)],  
 Apollo = Optional[Apollo(2600 g)],  
 Gac = Optional[Gac(3000 g)],  
 Hemi = Optional[Hemi(2600 g)],  
 Horned = Optional[Horned(1700 g)]  
}Copy

The minBy() and maxBy() collectors take a Comparator as an argument. In these examples, we have used the built-in Comparator.comparingInt​() function. Starting with JDK 8, the java.util.Comparator class was enriched with several new comparators, including the thenComparing() flavors for chaining comparators.

The issue here is represented by the optionals that should be removed. More generally, this category of issues continues to adapt the result returned by a collector to a different type.

Well, especially for these kinds of tasks, we have the collectingAndThen​(Collector<T,​A,​R> downstream, Function<R,​RR> finisher) factory method. This method takes a function that will be applied to the final result of the downstream collector (finisher). It can be used as follows:

Map<String, Integer> minMelonByType = melons.stream()  
 .collect(groupingBy(Melon::getType,  
 collectingAndThen(minBy(comparingInt(Melon::getWeight)),  
 m -> m.orElseThrow().getWeight())));Copy

The output will be as follows:

{Crenshaw=1200, Apollo=2600, Gac=1200, Hemi=1600, Horned=1700}Copy

We can also use maxMelonByType():

Map<String, Integer> maxMelonByType = melons.stream()  
 .collect(groupingBy(Melon::getType,   
 collectingAndThen(maxBy(comparingInt(Melon::getWeight)),  
 m -> m.orElseThrow().getWeight())));Copy

The output will be as follows:

{Crenshaw=1200, Apollo=2600, Gac=3000, Hemi=2600, Horned=1700}Copy

We may also want to group melons by type in Map<String, Melon[]>. Again, we can rely on collectingAndThen() for this, as follows:

Map<String, Melon[]> byTypeArray = melons.stream()  
 .collect(groupingBy(Melon::getType, collectingAndThen(  
 Collectors.toList(), l -> l.toArray(Melon[]::new))));Copy

Alternatively, we can create a generic collector and call it, as follows:

private static <T> Collector<T, ? , T[]>   
 toArray(IntFunction<T[]> func) {  
  
 return Collectors.collectingAndThen(  
 Collectors.toList(), l -> l.toArray(func.apply(l.size())));  
}  
  
Map<String, Melon[]> byTypeArray = melons.stream()  
 .collect(groupingBy(Melon::getType, toArray(Melon[]::new)));Copy

Multilevel grouping

Earlier, we mentioned that two of three flavors of groupingBy() take another collector as an argument. Moreover, we said that this can be any collector. By any collector, we mean groupingBy() as well.

By passing groupingBy() to groupingBy(), we can achieve *n*-levels of grouping or multilevel grouping. Mainly, we have *n*-levels of classification functions.

Let's consider the following list of Melon:

List<Melon> melonsSugar = Arrays.asList(  
 new Melon("Crenshaw", 1200, HIGH),  
 new Melon("Gac", 3000, LOW), new Melon("Hemi", 2600, HIGH),  
 new Melon("Hemi", 1600), new Melon("Gac", 1200, LOW),  
 new Melon("Cantaloupe", 2600, MEDIUM),  
 new Melon("Cantaloupe", 3600, MEDIUM),  
 new Melon("Apollo", 2600, MEDIUM), new Melon("Horned", 1200, HIGH),  
 new Melon("Gac", 3000, LOW), new Melon("Hemi", 2600, HIGH));Copy

So, each Melon has a type, a weight, and an indicator of sugar level. First, we want to group melons by the sugar indicator (LOW, MEDIUM, HIGH, or UNKNOWN (default)). Furthermore, we want to group melons by weight. This can be accomplished via two levels of grouping, as follows:

Map<Sugar, Map<Integer, Set<String>>> bySugarAndWeight = melonsSugar.stream()  
 .collect(groupingBy(Melon::getSugar,  
 groupingBy(Melon::getWeight, TreeMap::new,  
 mapping(Melon::getType, toSet()))));Copy

The output is as follows:

{  
 MEDIUM = {  
 2600 = [Apollo, Cantaloupe], 3600 = [Cantaloupe]  
 },  
 HIGH = {  
 1200 = [Crenshaw, Horned], 2600 = [Hemi]  
 },  
 UNKNOWN = {  
 1600 = [Hemi]  
 },  
 LOW = {  
 1200 = [Gac], 3000 = [Gac]  
 }  
}Copy

We can now say that Crenshaw and Horned weigh 1,200 g and have a high percentage of sugar. We also have Hemi at 2,600 g with a high percentage of sugar.

We can even represent our data in a table, as shown in the following diagram:

Tabla

Descripción generada automáticamente

Now, let's learn about partitioning.

190. Partitioning

Partitioning is a type of grouping that relies on a Predicate to divide a stream into two groups (a group for true and a group for false). The group for true stores the elements of the stream that have passed the predicate, while the group of false stores the rest of the elements (the elements that fail the predicate).

This Predicate represents the *classification function* of partitioning and is known as the *partitioning function*. Since the Predicate is evaluated to a boolean value, the partitioning operation returns a Map<Boolean, V>.

Let's assume that we have the following Melon class and List of Melon:

public class Melon {  
  
 private final String type;  
 private int weight;  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}  
  
List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 1200),  
 new Melon("Gac", 3000), new Melon("Hemi", 2600),  
 new Melon("Hemi", 1600), new Melon("Gac", 1200),  
 new Melon("Apollo", 2600), new Melon("Horned", 1700),  
 new Melon("Gac", 3000), new Melon("Hemi", 2600));Copy

Partitioning is done via Collectors.partitioningBy​(). This method comes in two flavors, and one of them receives a single argument, that is, partitioningBy​(Predicate<? super T> predicate).

For example, partitioning melons by a weight of 2,000 g with duplicates can be done as follows:

Map<Boolean, List<Melon>> byWeight = melons.stream()  
 .collect(partitioningBy(m -> m.getWeight() > 2000));Copy

The output will be as follows:

{  
 false=[Crenshaw(1200g),Hemi(1600g), Gac(1200g),Horned(1700g)],  
 true=[Gac(3000g),Hemi(2600g),Apollo(2600g), Gac(3000g),Hemi(2600g)]  
}Copy

The advantage of partitioning over filtering consists of the fact that partitioning keeps both lists of the stream elements.

The following diagram depicts how partitioningBy() works internally:

Diagrama

Descripción generada automáticamente

If we want to reject duplicates, then we can rely on other flavors of partitioningBy(), such as partitioningBy​(Predicate<? super T> predicate, Collector<? super T,​A,​D> downstream). The second argument allows us to specify another Collector for implementing the downstream reduction:

Map<Boolean, Set<Melon>> byWeight = melons.stream()  
 .collect(partitioningBy(m -> m.getWeight() > 2000, toSet()));Copy

The output will not contain duplicates:

{  
 false=[Horned(1700g), Gac(1200g), Crenshaw(1200g), Hemi(1600g)],   
 true=[Gac(3000g), Hemi(2600g), Apollo(2600g)]  
}Copy

Of course, in this case, distinct() will do the job as well:

Map<Boolean, List<Melon>> byWeight = melons.stream()  
 .distinct()  
 .collect(partitioningBy(m -> m.getWeight() > 2000));Copy

Other collectors can be used as well. For example, we can count the elements from each of these two groups via counting():

Map<Boolean, Long> byWeightAndCount = melons.stream()  
 .collect(partitioningBy(m -> m.getWeight() > 2000, counting()));Copy

The output will be as follows:

{false=4, true=5}Copy

We can also count the elements without duplicates:

Map<Boolean, Long> byWeight = melons.stream()  
 .distinct()  
 .collect(partitioningBy(m -> m.getWeight() > 2000, counting()));Copy

This time, the output will be as follows:

{false=4, true=3}Copy

Finally, partitioningBy() can be combined with collectingAndThen(), which we introduced in the *Grouping* section. For example, let's partition the melons by weight of 2,000 g and keep only the heaviest from each partition:

Map<Boolean, Melon> byWeightMax = melons.stream()  
 .collect(partitioningBy(m -> m.getWeight() > 2000,   
 collectingAndThen(maxBy(comparingInt(Melon::getWeight)),  
 Optional::get)));Copy

The output will be as follows:

{false=Horned(1700g), true=Gac(3000g)}Copy

191. Filtering, flattening, and mapping collectors

Let's assume that we have the following Melon class and List of Melon:

public class Melon {  
  
 private final String type;  
 private final int weight;  
 private final List<String> pests;  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}  
  
List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 2000),  
 new Melon("Hemi", 1600), new Melon("Gac", 3000),  
 new Melon("Hemi", 2000), new Melon("Crenshaw", 1700),  
 new Melon("Gac", 3000), new Melon("Hemi", 2600));Copy

The Java Stream API provides filtering(), flatMapping(), and mapping(), especially for use in multi-level reductions (such as the downstream of groupingBy() or partitioningBy()).

Conceptually, the goal of filtering() is the same as filter(), the goal of flatMapping() is the same as flatMap(), and the goal of mapping() is the as map().

filtering()

User problem: *I want to take all the melons that are heavier than 2,000 g and group them by their type. For each type, add them to the proper container (there is a container for each type – just check the container's labels).*

By using filtering​(Predicate<? super T> predicate, Collector<? super T,​A,​R> downstream), we apply a predicate to each element of the current collector and accumulate the output in the downstream collector.

So, to group the melons that are heavier than 2,000 g by type, we can write the following stream pipeline:

Map<String, Set<Melon>> melonsFiltering = melons.stream()  
 .collect(groupingBy(Melon::getType,  
 filtering(m -> m.getWeight() > 2000, toSet())));Copy

The output will be as follows (each Set<Melon> is a container):

{Crenshaw=[], Gac=[Gac(3000g)], Hemi=[Hemi(2600g)]}Copy

Notice that there is no Crenshaw heavier than 2,000 g, so filtering() has mapped this type to an empty set (container). Now, let's rewrite this via filter():

Map<String, Set<Melon>> melonsFiltering = melons.stream()  
 .filter(m -> m.getWeight() > 2000)  
 .collect(groupingBy(Melon::getType, toSet()));Copy

Because filter() doesn't perform mappings for elements that fail its predicate, the output will look as follows:

{Gac=[Gac(3000g)], Hemi=[Hemi(2600g)]}Copy

User problem: *This time, I am interested only in the melons of the Hemi type. There are two containers: one for Hemi melons lighter than (or equal to) 2,000 g and one for Hemi melons heavier than 2,000 g.*

Filtering can be used with partitioningBy() as well. To partition melons heavier than 2,000 g and filter them by a certain type (in this case, Hemi), we have the following:

Map<Boolean, Set<Melon>> melonsFiltering = melons.stream()  
 .collect(partitioningBy(m -> m.getWeight() > 2000,  
 filtering(m -> m.getType().equals("Hemi"), toSet())));Copy

The output is as follows:

{false=[Hemi(1600g), Hemi(2000g)], true=[Hemi(2600g)]}Copy

Applying filter() will lead to the same result:

Map<Boolean, Set<Melon>> melonsFiltering = melons.stream()  
 .filter(m -> m.getType().equals("Hemi"))  
 .collect(partitioningBy(m -> m.getWeight() > 2000, toSet()));Copy

The output is as follows:

{false=[Hemi(1600g), Hemi(2000g)], true=[Hemi(2600g)]}Copy

mapping()

User problem: *For each type of melon, I want the list of weights in ascending order.*

By using mapping​(Function<? super T,​? extends U> mapper, Collector<? super U,​A,​R> downstream), we can apply a mapping function to each element of the current collector and accumulate the output in the downstream collector.

For example, for grouping the weights of melons by type, we can write the following snippet of code:

Map<String, TreeSet<Integer>> melonsMapping = melons.stream()  
 .collect(groupingBy(Melon::getType,  
 mapping(Melon::getWeight, toCollection(TreeSet::new))));Copy

The output will be as follows:

{Crenshaw=[1700, 2000], Gac=[3000], Hemi=[1600, 2000, 2600]}Copy

User problem: *I want two lists. One should contain the melon types lighter than (or equal to) 2,000 g and the other one should contain the rest of the types.*

Partitioning melons that are heavier than 2,000 g and collecting only their types can be done as follows:

Map<Boolean, Set<String>> melonsMapping = melons.stream()  
 .collect(partitioningBy(m -> m.getWeight() > 2000,  
 mapping(Melon::getType, toSet())));Copy

The output is as follows:

{false=[Crenshaw, Hemi], true=[Gac, Hemi]}Copy

flatMapping()

For a quick reminder about flattening a stream, it is advisable to read the *Map a stream* section.

Now, let's assume that we have the following list of Melon (notice that we've added the names of pests as well):

List<Melon> melonsGrown = Arrays.asList(  
 new Melon("Honeydew", 5600,  
 Arrays.asList("Spider Mites", "Melon Aphids", "Squash Bugs")),  
 new Melon("Crenshaw", 2000,  
 Arrays.asList("Pickleworms")),  
 new Melon("Crenshaw", 1000,  
 Arrays.asList("Cucumber Beetles", "Melon Aphids")),  
 new Melon("Gac", 4000,  
 Arrays.asList("Spider Mites", "Cucumber Beetles")),  
 new Melon("Gac", 1000,  
 Arrays.asList("Squash Bugs", "Squash Vine Borers")));Copy

User problem: *For each type of melon, I want a list of their pests.*

So, let's group melons by type and collect their pests. Each melon has none, one, or multiple pests, and so we expect an output of the Map<String, List<String>> type. The first attempt will rely on mapping():

Map<String, List<List<String>>> pests = melonsGrown.stream()  
 .collect(groupingBy(Melon::getType,   
 mapping(m -> m.getPests(), toList())));Copy

Obviously, this is not a good approach since the returned type is Map<String, List<List<String>>>.

Another naive approach that relies on mapping is as follows:

Map<String, List<List<String>>> pests = melonsGrown.stream()  
 .collect(groupingBy(Melon::getType,   
 mapping(m -> m.getPests().stream(), toList())));Copy

Obviously, this is not a good approach either since the returned type is Map<String, List<Stream<String>>>.

It's time to introduce flatMapping(). By using flatMapping​(Function<? super T,​? extends Stream<? extends U>> mapper, Collector<? super U,​A,​R> downstream), we apply the flatMapping function to each element of the current collector and accumulate the output in the downstream collector:

Map<String, Set<String>> pestsFlatMapping = melonsGrown.stream()  
 .collect(groupingBy(Melon::getType,   
 flatMapping(m -> m.getPests().stream(), toSet())));Copy

This time, the type looks fine and the output is as follows:

{  
 Crenshaw = [Cucumber Beetles, Pickleworms, Melon Aphids],  
 Gac = [Cucumber Beetles, Squash Bugs, Spider Mites,   
 Squash Vine Borers],  
 Honeydew = [Squash Bugs, Spider Mites, Melon Aphids]  
}Copy

User problem: *I want two lists. One should contain the pests of melons lighter than 2,000 g, and the other should contain the pests of the rest of melons.*

Partitioning melons heavier than 2,000 g and collecting the pests can be done as follows:

Map<Boolean, Set<String>> pestsFlatMapping = melonsGrown.stream()  
 .collect(partitioningBy(m -> m.getWeight() > 2000,   
 flatMapping(m -> m.getPests().stream(), toSet())));Copy

The output is as follows:

{  
 false = [Cucumber Beetles, Squash Bugs, Pickleworms, Melon Aphids,  
 Squash Vine Borers],  
 true = [Squash Bugs, Cucumber Beetles, Spider Mites, Melon Aphids]  
}Copy

192. Teeing

Starting with JDK 12, we can merge the results of two collectors via Collectors.teeing():

* public static <T,​R1,​R2,​R> Collector<T,​?,​R> teeing​(Collector<? super T,​?,​R1> downstream1, Collector<? super T,​?,​R2> downstream2, BiFunction<? super R1,​? super R2,​R> merger):



The result is a Collector that is a composite of two passed downstream collectors. Every element that's passed to the resulting collector is processed by both downstream collectors, and then their results are merged into the final result using the specified BiFunction.

Let's take a look at a classical problem. The following class simply stores the number of elements in a stream of integers and their sum:

public class CountSum {  
  
 private final Long count;  
 private final Integer sum;  
  
 public CountSum(Long count, Integer sum) {  
 this.count = count;  
 this.sum = sum;  
 }  
 ...  
}Copy

We can obtain this information via teeing(), as follows:

CountSum countsum = Stream.of(2, 11, 1, 5, 7, 8, 12)  
 .collect(Collectors.teeing(  
 counting(),  
 summingInt(e -> e),  
 CountSum::new));Copy

Here, we have applied two collectors to each element from the stream (counting() and summingInt()) and the results have been merged in an instance of CountSum:

CountSum{count=7, sum=46}Copy

Let's take a look at another problem. This time, the MinMax class stores the minimum and maximum of a stream of integers:

public class MinMax {  
  
 private final Integer min;  
 private final Integer max;  
  
 public MinMax(Integer min, Integer max) {  
 this.min = min;  
 this.max = max;  
 }  
 ...  
}Copy

Now, we can obtain this information like so:

MinMax minmax = Stream.of(2, 11, 1, 5, 7, 8, 12)  
 .collect(Collectors.teeing(  
 minBy(Comparator.naturalOrder()),  
 maxBy(Comparator.naturalOrder()),  
 (Optional<Integer> a, Optional<Integer> b)   
 -> new MinMax(a.orElse(Integer.MIN\_VALUE),  
 b.orElse(Integer.MAX\_VALUE))));Copy

Here, we have applied two collectors to each element from the stream (minBy() and maxBy()) and the results have been merged in an instance of MinMax:

MinMax{min=1, max=12}Copy

Finally, let's consider the following Melon class and List of Melon:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 public Melon(String type, int weight) {  
 this.type = type;  
 this.weight = weight;  
 }  
 ...  
}  
  
List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 1200),  
 new Melon("Gac", 3000), new Melon("Hemi", 2600),  
 new Melon("Hemi", 1600), new Melon("Gac", 1200),  
 new Melon("Apollo", 2600), new Melon("Horned", 1700),  
 new Melon("Gac", 3000), new Melon("Hemi", 2600));Copy

The aim here is to compute the total weight of these melons and list their weights. We can map this as follows:

public class WeightsAndTotal {  
  
 private final int totalWeight;  
 private final List<Integer> weights;  
  
 public WeightsAndTotal(int totalWeight, List<Integer> weights) {  
 this.totalWeight = totalWeight;  
 this.weights = weights;  
 }  
 ...  
}Copy

The solution to this problem relies on Collectors.teeing(), as follows:

WeightsAndTotal weightsAndTotal = melons.stream()  
 .collect(Collectors.teeing(  
 summingInt(Melon::getWeight),  
 mapping(m -> m.getWeight(), toList()),  
 WeightsAndTotal::new));Copy

This time, we have applied the summingInt() and mapping() collectors. The output is as follows:

WeightsAndTotal {  
 totalWeight = 19500,  
 weights = [1200, 3000, 2600, 1600, 1200, 2600, 1700, 3000, 2600]  
}Copy

193. Writing a custom collector

Let's assume that we have the following Melon class and List of Melon:

public class Melon {  
  
 private final String type;  
 private final int weight;  
 private final List<String> grown;  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}  
  
List<Melon> melons = Arrays.asList(new Melon("Crenshaw", 1200),  
 new Melon("Gac", 3000), new Melon("Hemi", 2600),  
 new Melon("Hemi", 1600), new Melon("Gac", 1200),  
 new Melon("Apollo", 2600), new Melon("Horned", 1700),  
 new Melon("Gac", 3000), new Melon("Hemi", 2600));Copy

In the *Partitioning* section, we saw how to use the partitioningBy() collector to partition melons that weigh 2,000 g with duplicates:

Map<Boolean, List<Melon>> byWeight = melons.stream()  
 .collect(partitioningBy(m -> m.getWeight() > 2000));Copy

Now, let's see if we can achieve the same result via a dedicated custom collector.

Let's begin by saying that writing a custom collector is not a day-to-day task, but it may be useful to know how to do it. The built-in Java Collector interface looks as follows:

public interface Collector<T, A, R> {  
 Supplier<A> supplier();  
 BiConsumer<A, T> accumulator();  
 BinaryOperator<A> combiner();  
 Function<A, R> finisher();  
 Set<Characteristics> characteristics();  
 ...  
}Copy

To write a custom collector, it is very important to know that T, A, and R represent the following:

* T represents the type of elements from the Stream (elements that will be collected).
* A represents the type of object that was used during the collection process known as the *accumulator*, which is used to accumulate the stream elements in a *mutable result container.*
* R represents the type of the object after the collection process (the final result).

A collector may return the accumulator itself as the final result or may perform an optional transformation on the accumulator to obtain the final result (perform an optional final transformation from the intermediate accumulation type, A,to the final result type, R).

In terms of our problem, we know that T is Melon, A is Map<Boolean, List<Melon>>, and R is Map<Boolean, List<Melon>>. This collector returns the accumulator itself as the final result via Function.identity(). That being said, we can start our custom collector as follows:

public class MelonCollector implements  
 Collector<Melon, Map<Boolean, List<Melon>>,  
 Map<Boolean, List<Melon>>> {  
 ...  
}Copy

So, a Collector is specified by four functions. These functions are working together to accumulate entries into a mutable result container, and optionally perform a final transformation on the result. They are as follows:

* Creating a new empty mutable result container (supplier())
* Incorporating a new data element into the mutable result container (accumulator())
* Combining two mutable result containers into one (combiner())
* Performing an optional final transformation on the mutable result container to obtain the final result (finisher())

In addition, the behavior of the collector is defined in the last method, characteristics(). Set<Characteristics> may contain the following four values:

* UNORDERED: The order of accumulating/collecting elements is not important for the final result.
* CONCURRENT: The elements of the stream can be accumulated by multiple threads in a concurrent fashion (in the end, the collector can perform a parallel reduction of the stream. The containers resulting from the parallel processing of the stream are combined in a single result container. The source of data should be unordered by its nature or the UNORDERED flag should be present.
* IDENTITY\_FINISH: Indicates that the accumulator itself is the final result (basically, we can cast A to R); in this case, the finisher() is not called.

The supplier – Supplier<A> supplier();

The job of supplier() is to return (at every call) a Supplier of an empty mutable result container.

In our case, the result container is of the Map<Boolean, List<Melon>> type, and so supplier() can be implemented as follows:

@Override  
public Supplier<Map<Boolean, List<Melon>>> supplier() {  
  
 return () -> {  
 return new HashMap<Boolean, List<Melon>> () {  
 {  
 put(true, new ArrayList<>());  
 put(false, new ArrayList<>());  
 }  
 };  
 };  
}Copy

In parallel execution, this method may be called multiple times.

Accumulating elements – BiConsumer<A, T> accumulator();

The accumulator() method returns the function that performs the reduction operation. This is BiConsumer, which is an operation that accepts two input arguments and returns no result. The first input argument is the current result container (being the result of the reduction so far), and the second input argument is the current element from the stream. This function modifies the result container itself by accumulating the traversed element or an effect of traversing this element. In our case, accumulator() adds the currently traversed element to one of the two ArrayLists:

@Override  
public BiConsumer<Map<Boolean, List<Melon>>, Melon> accumulator() {  
  
 return (var acc, var melon) -> {  
 acc.get(melon.getWeight() > 2000).add(melon);  
 };  
}Copy

Applying the final transformation – Function<A, R> finisher();

The finisher() method returns a function that is applied at the end of the accumulation process. When this method is invoked, there are no more stream elements to traverse. All of the elements will be accumulated transformed from the intermediate accumulation type, A, to the final result type, R. If no transformation is needed, then we can return the intermediate result (the accumulator itself):

@Override  
public Function<Map<Boolean, List<Melon>>,  
 Map<Boolean, List<Melon>>> finisher() {  
  
 return Function.identity();  
}Copy

Parallelizing the collector – BinaryOperator<A> combiner();

If the stream is processed in parallel, then different threads (accumulators) will produce partial result containers. In the end, these partial results must be merged in a single one. This is exactly what combiner() does. In this case, the combiner() method needs to merge two maps by adding all the values from the two lists of the second Map to the corresponding lists in the first Map:

@Override  
public BinaryOperator<Map<Boolean, List<Melon>>> combiner() {  
  
 return (var map, var addMap) -> {  
 map.get(true).addAll(addMap.get(true));  
 map.get(false).addAll(addMap.get(false));  
  
 return map;  
 };  
}Copy

Returning the final result – Function<A, R> finisher();

The final result is computed in the finisher() method. In this case, we simply return Function.identity() since the accumulator doesn't require any further transformation:

@Override  
public Function<Map<Boolean, List<Melon>>,  
 Map<Boolean, List<Melon>>> finisher() {  
  
 return Function.identity();  
}Copy

Characteristics – Set<Characteristics> characteristics();

Finally, we indicate that our collector is IDENTITY\_FINISH and CONCURRENT:

@Override  
public Set<Characteristics> characteristics() {  
 return Set.of(IDENTITY\_FINISH, CONCURRENT);  
}Copy

The code that's bundled with this book glues all the pieces of the puzzle together in a class named MelonCollector.

Testing time

MelonCollector can be used via the new keyword, as follows:

Map<Boolean, List<Melon>> melons2000 = melons.stream()  
 .collect(new MelonCollector());Copy

We will receive the following output:

{  
 false = [Crenshaw(1200 g),Hemi(1600 g),Gac(1200 g),Horned(1700 g)],  
 true = [Gac(3000 g),Hemi(2600 g),Apollo(2600 g),  
 Gac(3000 g),Hemi(2600 g)]  
}Copy

We can also use it via parallelStream():

Map<Boolean, List<Melon>> melons2000 = melons.parallelStream()  
 .collect(new MelonCollector());Copy

If we use the combiner() method, then the output may look as follows:

{false = [], true = [Hemi(2600g)]}   
 ForkJoinPool.commonPool - worker - 7  
...  
{false = [Horned(1700g)], true = []}   
 ForkJoinPool.commonPool - worker - 15   
{false = [Crenshaw(1200g)], true = [Gac(3000g)]}   
 ForkJoinPool.commonPool - worker - 9  
...  
{false = [Crenshaw(1200g), Hemi(1600g), Gac(1200g), Horned(1700g)],   
true = [Gac(3000g), Hemi(2600g), Apollo(2600g),   
 Gac(3000g), Hemi(2600g)]}Copy

Custom collecting via collect()

In the case of an IDENTITY\_FINISH collection operation, there is at least one more solution for obtaining a custom collector. This solution is facilitated by the following method:

<R> R collect​(Supplier<R> supplier, BiConsumer<R,​? super T> accumulator, BiConsumer<R,​R> combiner)Copy

This flavor of collect() is a great fit as long as we deal with an IDENTITY\_FINISH collection operation and we can provide a supplier, accumulator, and combiner.

Let's take a look at some examples:

List<String> numbersList = Stream.of("One", "Two", "Three")  
 .collect(ArrayList::new, ArrayList::add,  
 ArrayList::addAll);  
  
Deque<String> numbersDeque = Stream.of("One", "Two", "Three")  
 .collect(ArrayDeque::new, ArrayDeque::add,  
 ArrayDeque::addAll);  
  
String numbersString = Stream.of("One", "Two", "Three")  
 .collect(StringBuilder::new, StringBuilder::append,  
 StringBuilder::append).toString();Copy

You can use these examples to identify more JDK classes whose signatures are well-suited for use with method references as arguments to collect().

194. Method reference

Let's assume that we have the following Melon class and List of Melon:

public class Melon {  
  
 private final String type;  
 private int weight;  
  
 public static int growing100g(Melon melon) {  
 melon.setWeight(melon.getWeight() + 100);  
  
 return melon.getWeight();  
 }  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}  
  
List<Melon> melons = Arrays.asList(  
 new Melon("Crenshaw", 1200), new Melon("Gac", 3000),  
 new Melon("Hemi", 2600), new Melon("Hemi", 1600));Copy

In a nutshell, *method references* are shortcuts for lambda expressions.

Mainly, method reference is a technique that's used to call a method by name rather than by a description of how to call it. The main benefit is readability.

A method reference is written by placing the target reference before the delimiter, ::, and the name of the method is provided after it.

We'll take a look at all four kinds of method references in the upcoming sections.

Method reference to a static method

We can group each Melon from the aforementioned list that's 100 g via the static method called growing100g():

* No method reference:

melons.forEach(m -> Melon.growing100g(m));Copy

* Method reference:

melons.forEach(Melon::growing100g);Copy

Method reference to an instance method

Let's assume that we are defining the following Comparator for Melon:

public class MelonComparator implements Comparator {  
  
 @Override  
 public int compare(Object m1, Object m2) {  
 return Integer.compare(((Melon) m1).getWeight(),  
 ((Melon) m2).getWeight());  
 }  
}Copy

Now, we can refer to it as follows:

* No method reference:

MelonComparator mc = new MelonComparator();  
  
List<Melon> sorted = melons.stream()  
 .sorted((Melon m1, Melon m2) -> mc.compare(m1, m2))  
 .collect(Collectors.toList());Copy

* Method reference:

List<Melon> sorted = melons.stream()  
 .sorted(mc::compare)  
 .collect(Collectors.toList());Copy

Of course, we can call Integer.compare() directly as well:

* No method reference:

List<Integer> sorted = melons.stream()  
 .map(m -> m.getWeight())  
 .sorted((m1, m2) -> Integer.compare(m1, m2))  
 .collect(Collectors.toList());Copy

* Method reference:

List<Integer> sorted = melons.stream()  
 .map(m -> m.getWeight())  
 .sorted(Integer::compare)  
 .collect(Collectors.toList());Copy

Method reference to a constructor

Referring a constructor can be done via the new keyword, as follows:

BiFunction<String, Integer, Melon> melonFactory = Melon::new;  
Melon hemi1300 = melonFactory.apply("Hemi", 1300);Copy

More details and examples about method reference to a constructor are available in the *Implementing the factory pattern* section in the previous chapter.

195. Parallel processing of streams

In a nutshell, parallel processing a stream refers to a process that consists of three steps:

1. Splitting the elements of a stream into multiple chunks
2. Processing each chunk in a separate thread
3. Joining the results of processing in a single result

These three steps take place behind the scenes via the default ForkJoinPool method as we discussed in [Chapter 10](https://subscription.packtpub.com/book/programming/9781789801415/9/ch09lvl1sec36/cb1d3156-ee13-4790-9bac-81d08d59d7d2.xhtml), *Concurrency – Thread Pools, Callables, and Synchronizers* and [Chapter 11](https://subscription.packtpub.com/book/programming/9781789801415/9/ch09lvl1sec36/b466dd92-c030-4835-b259-fdf24f4a36b1.xhtml), *Concurrency – Deep Dive*.

As a rule of thumb, parallel processing can be applied only to *stateless* (the state of an element doesn't affect another element), *non-interfering* (the data source is not affected), and *associative* (the result is not affected by the order of operands) operations.

Let's assume that our problem is to sum the elements of a list of doubles:

Random rnd = new Random();  
List<Double> numbers = new ArrayList<>();  
  
for (int i = 0; i < 1 \_000\_000; i++) {  
 numbers.add(rnd.nextDouble());  
}Copy

We can also do this directly as a stream:

DoubleStream.generate(() -> rnd.nextDouble()).limit(1\_000\_000)Copy

In a sequential approach, we can do this as follows:

double result = numbers.stream()  
 .reduce((a, b) -> a + b).orElse(-1d);Copy

This operation will probably take place on a single core behind the scenes (even if our machine has more cores), as shown in the following diagram:

Imagen que contiene Código QR

Descripción generada automáticamente

This problem is a good candidate for leverage parallelization, and so we can call parallelStream() instead of stream(), as follows:

double result = numbers.parallelStream()  
 .reduce((a, b) -> a + b).orElse(-1d);Copy

Once we call parallelStream(), Java will take action and process the stream using multiple threads. Parallelization can be done via the parallel() method as well:

double result = numbers.stream()  
 .parallel()  
 .reduce((a, b) -> a + b).orElse(-1d);Copy

This time, the processing takes place via a fork/join, as shown in the following diagram (there is one thread for each available core):

Diagrama

Descripción generada automáticamente

In the context of reduce(), parallelization can be depicted as follows:

Diagrama

Descripción generada automáticamente

By default, the Java ForkJoinPool will try to fetch as many threads as available processors like so:

int noOfProcessors = Runtime.getRuntime().availableProcessors();Copy

We can affect the number of threads globally (all the parallel streams will use it) as follows:

System.setProperty(  
 "java.util.concurrent.ForkJoinPool.common.parallelism", "10");Copy

Alternatively, we can affect the number of threads for a single parallel stream as follows:

ForkJoinPool customThreadPool = new ForkJoinPool(5);  
  
double result = customThreadPool.submit(  
 () -> numbers.parallelStream()  
 .reduce((a, b) -> a + b)).get().orElse(-1d);Copy

Affecting the number of threads is an important decision to make. Trying to determine the optimal number of threads depending on the environment is not an easy task and, in most scenarios, the default setting (*number of threads = number of processors*) is the most suitable.

Even if the problem is a good candidate for leverage parallelization, it doesn't mean that parallel processing is a silver bullet. Deciding to go with parallel processing or not should be a decision that's made after benchmarking and comparing sequential versus parallel processing. Most commonly, parallel processing acts better in the case of huge datasets.

Do not fall into the trap of thinking that a larger number of threads results in faster processing. Avoid something like the following (these numbers are just indicators for a machine with 8 cores):

5 threads (~40 ms)  
20 threads (~50 ms)  
100 threads (~70 ms)  
1000 threads (~ 250 ms)Copy

Spliterators

A Java Spliterator interface (also known as a *splittable iterator*) is an interface that's used to traverse the elements of a source (for example, a collection or stream) in parallel. This interface defines the following methods:

public interface Spliterator<T> {  
 boolean tryAdvance(Consumer<? super T> action);  
 Spliterator<T> trySplit();  
 long estimateSize();  
 int characteristics();  
}Copy

Let's consider a simple list of 10 integers:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);Copy

We can obtain a Spliterator interface for this list like so:

Spliterator<Integer> s1 = numbers.spliterator();Copy

We can also do the same from a stream:

Spliterator<Integer> s1 = numbers.stream().spliterator();Copy

In order to advance to (traverse) the first element, we need to call the tryAdvance() method, as follows:

s1.tryAdvance(e   
 -> System.out.println("Advancing to the   
 first element of s1: " + e));Copy

We will receive the following output:

Advancing to the first element of s1: 1Copy

Spliterator can try to estimate the number of elements left to traverse via the estimateSize() method, as follows:

System.out.println("\nEstimated size of s1: " + s1.estimateSize());Copy

We will receive the following output (we've traversed one element; there are nine to go):

Estimated size of s1: 9Copy

We can split this into two via a Spliterator interface using the trySplit() method. The result will be another Spliterator interface:

Spliterator<Integer> s2 = s1.trySplit();Copy

Checking the number of elements reveals the effect of trySplit():

System.out.println("Estimated size s1: " + s1.estimateSize());  
System.out.println("Estimated size s2: " + s2.estimateSize());Copy

We will receive the following output:

Estimated size s1: 5  
Estimated size s2: 4Copy

Trying to print all the elements from s1 and s2 can be accomplished using forEachRemaining(), as follows:

s1.forEachRemaining(System.out::println); // 6, 7, 8, 9, 10  
s2.forEachRemaining(System.out::println); // 2, 3, 4, 5Copy

A Spliterator interface defines a suite of constants for its characteristics – CONCURRENT (4096), DISTINCT (1), IMMUTABLE (1024), NONNULL (256), ORDERED (16), SIZED (64), SORTED (4), and SUBSIZED (16384).

We can print the characteristics via the characteristics() method as follows:

System.out.println(s1.characteristics()); // 16464  
System.out.println(s2.characteristics()); // 16464Copy

It is simpler to test whether a certain characteristic is presented using hasCharacteristics():

if (s1.hasCharacteristics(Spliterator.ORDERED)) {  
 System.out.println("ORDERED");  
}  
  
if (s1.hasCharacteristics(Spliterator.SIZED)) {  
 System.out.println("SIZED");  
}Copy

Writing a custom Spliterator

Obviously, writing a custom Spliterator is not a daily task, but let's assume that we are working on a project that, for some reason, requires us to process strings that contain ideographic characters (**CJKV** (short for **Chinese**, **Japanese**, **Korean**, and **Vietnamese**)) and non-ideographic characters. We want to process these strings in parallel. This mandates that we split them into characters only at positions representing ideographic characters.

Obviously, the default Spliterator will not perform as we want it to, and so we may need to write a custom Spliterator. For this, we have to implement the Spliterator interface and provide an implementation of a few methods. The implementation is available in the code that's been bundled with this book. Consider opening the IdeographicSpliterator source code and keeping it close by while reading the rest of this section.

The climax of the implementation is in the trySplit() method. Here, we are trying to split the current string in half and continue to traverse it until we find an ideographic character. For checking purposes, we've just added the following line:

System.out.println("Split successfully at character: "   
 + str.charAt(splitPosition));Copy

Now, let's consider a string containing ideographic characters:

String str = "Character Information Forma

Descripción generada automáticamente con confianza baja Development and Maintenance "   
 + "Project Forma

Descripción generada automáticamente con confianza baja for e-Government MojiJoho-Kiban Forma

Descripción generada automáticamente con confianza baja Project";Copy

Now, let's create a parallel stream for this string and force IdeographicSpliterator to do its job:

Spliterator<Character> spliterator = new IdeographicSpliterator(str);  
Stream<Character> stream = StreamSupport.stream(spliterator, true);  
  
// force spliterator to do its job  
stream.collect(Collectors.toList());Copy

One possible output will reveal that the split takes place only at positions containing ideographic characters:

Split successfully at character: Forma

Descripción generada automáticamente con confianza baja  
Split successfully at character: Forma

Descripción generada automáticamente con confianza bajaCopy

196. Null-safe streams

The problem with creating a Stream of an element that may or may not be null can be solved using Optional.ofNullable() or, even better via JDK 9, Stream.ofNullable():

* static <T> Stream<T> ofNullable​(T t)

This method gets a single element (T) and returns a sequential Stream containing this single element (Stream<T>); otherwise, it returns an empty Stream if it's not null.

For example, we can write a helper method that wraps the call to Stream.ofNullable() as follows:

public static <T> Stream<T> elementAsStream(T element) {  
 return Stream.ofNullable(element);  
}Copy

If this method lives in a utility class named AsStreams, then we can perform several calls, as follows:

// 0  
System.out.println("Null element: "   
 + AsStreams.elementAsStream(null).count());  
  
// 1  
System.out.println("Non null element: "   
 + AsStreams.elementAsStream("Hello world").count());Copy

Notice that when we pass null, we get an empty stream (the count() method returns 0)!

If our element is a collection, then things become more interesting. For example, let's assume that we have the following list (notice that this list contains some null values):

List<Integer> ints = Arrays.asList(5, null, 6, null, 1, 2);Copy

Now, let's write a helper method that returns a Stream<T>, where T is a collection:

public static <T> Stream<T> collectionAsStreamWithNulls(  
 Collection<T> element) {  
 return Stream.ofNullable(element).flatMap(Collection::stream);  
}Copy

If we call this method with null, then we obtain an empty stream:

// 0  
System.out.println("Null collection: "   
 + AsStreams.collectionAsStreamWithNulls(null).count());Copy

Now, if we call it with our list, ints, then we obtain a Stream<Integer>:

// 6  
System.out.println("Non-null collection with nulls: "  
 + AsStreams.collectionAsStreamWithNulls(ints).count());Copy

Notice that the stream has six elements (all the elements from the underlying list)—5, null, 6, null, 1, and 2.

If we know that the collection itself is not null, but it may contain null values, then we can write another helper method, as follows:

public static <T> Stream<T> collectionAsStreamWithoutNulls(  
 Collection<T> collection) {  
  
 return collection.stream().flatMap(e -> Stream.ofNullable(e));  
}Copy

This time, if the collection itself is null, then the code will throw an NullPointerException. However, if we pass our list to it, then the result will be a Stream<Integer> without null values:

// 4  
System.out.println("Non-null collection without nulls: "   
 + AsStreams.collectionAsStreamWithoutNulls(ints).count());Copy

The returned stream has only four elements—5, 6, 1, and 2.

Finally, if the collection itself may be null and may contain null values, then the following helper will do the job and return a null-safe stream:

public static <T> Stream<T> collectionAsStream(  
 Collection<T> collection) {  
  
 return Stream.ofNullable(collection)  
 .flatMap(Collection::stream)  
 .flatMap(Stream::ofNullable);  
}Copy

If we pass null, then we get an empty stream:

// 0  
System.out.println(  
 "Null collection or non-null collection with nulls: "   
 + AsStreams.collectionAsStream(null).count());Copy

If we pass our list, we get a Stream<Integer> stream without null values:

// 4  
System.out.println(  
 "Null collection or non-null collection with nulls: "   
 + AsStreams.collectionAsStream(ints).count());Copy

197. Composing functions, predicates, and comparators

Composing (or chaining) functions, predicates, and comparators allows us to write compound criteria that should be applied in unison.

Composing predicates

Let's assume that we have the following Melon class and List of Melon:

public class Melon {  
  
 private final String type;  
 private final int weight;  
  
 // constructors, getters, setters, equals(),  
 // hashCode(), toString() omitted for brevity  
}  
  
List<Melon> melons = Arrays.asList(new Melon("Gac", 2000),  
 new Melon("Horned", 1600), new Melon("Apollo", 3000),  
 new Melon("Gac", 3000), new Melon("Hemi", 1600));Copy

The Predicate interface comes with three methods that take a Predicate and uses it to obtain an enriched Predicate. These methods are and(), or(), and negate().

For example, let's assume that we want to filter the melons that are heavier than 2,000 g. For this, we can write a Predicate, as follows:

Predicate<Melon> p2000 = m -> m.getWeight() > 2000;Copy

Now, let's assume that we want to enrich this Predicate to filter only melons that respect p2000 and are of the Gac or Apollo type. For this, we can use the and() and or() methods, as follows:

Predicate<Melon> p2000GacApollo   
 = p2000.and(m -> m.getType().equals("Gac"))  
 .or(m -> m.getType().equals("Apollo"));Copy

This is interpreted from left to rights as a && (b || c), where we have the following:

* a is m -> m.getWeight() > 2000
* b is m -> m.getType().equals("Gac")
* c is m -> m.getType().equals("Apollo")

Obviously, we can add more criteria in the same manner.

Let's pass this Predicate to filter():

// Apollo(3000g), Gac(3000g)  
List<Melon> result = melons.stream()  
 .filter(p2000GacApollo)  
 .collect(Collectors.toList());Copy

Now, let's assume that our problem requires that we obtain the negation of the aforementioned compound predicate. It is cumbersome to rewrite this predicate as !a && !b && !c or any other counterpart expression. A better solution is to call the negate() method, as follows:

Predicate<Melon> restOf = p2000GacApollo.negate();Copy

Let's pass it to filter():

// Gac(2000g), Horned(1600g), Hemi(1600g)  
List<Melon> result = melons.stream()  
 .filter(restOf)  
 .collect(Collectors.toList());Copy

Starting with JDK 11, we can negate a Predicate that's passed as an argument to the not() method. For example, let's filter all the melons that are lighter than (or equal to) 2,000 g using not():

Predicate<Melon> pNot2000 = Predicate.not(m -> m.getWeight() > 2000);  
  
// Gac(2000g), Horned(1600g), Hemi(1600g)  
List<Melon> result = melons.stream()  
 .filter(pNot2000)  
 .collect(Collectors.toList());Copy

Composing comparators

Let's consider the same Melon class and List of Melon from the preceding section.

Now, let's sort this List of Melon by weight using Comparator.comparing():

Comparator<Melon> byWeight = Comparator.comparing(Melon::getWeight);  
  
// Horned(1600g), Hemi(1600g), Gac(2000g), Apollo(3000g), Gac(3000g)  
List<Melon> sortedMelons = melons.stream()  
 .sorted(byWeight)  
 .collect(Collectors.toList());Copy

We can sort the list by type as well:

Comparator<Melon> byType = Comparator.comparing(Melon::getType);  
  
// Apollo(3000g), Gac(2000g), Gac(3000g), Hemi(1600g), Horned(1600g)  
List<Melon> sortedMelons = melons.stream()  
 .sorted(byType)  
 .collect(Collectors.toList());Copy

To reverse the sorting order, simply call reversed():

Comparator<Melon> byWeight   
 = Comparator.comparing(Melon::getWeight).reversed();Copy

So far, so good!

Now, let's assume that we want to sort the list by weight and type. In other words, when two melons have the same weight (for example, Horned (1600g), Hemi(1600g)) they should be sorted by type (for example, Hemi(1600g), Horned(1600g)). A naive approach will look as follows:

// Apollo(3000g), Gac(2000g), Gac(3000g), Hemi(1600g), Horned(1600g)  
List<Melon> sortedMelons = melons.stream()  
 .sorted(byWeight)  
 .sorted(byType)  
 .collect(Collectors.toList());Copy

Obviously, the result is not what we expected. This is happening because the comparators have not been applied to the same list. The byWeight comparator is applied to the original list, while the byType comparator is applied to the output of byWeight. Basically, byType cancels the effect of byWeight.

The solution comes from the Comparator.thenComparing() method. This method allows us to chain comparators:

Comparator<Melon> byWeightAndType   
 = Comparator.comparing(Melon::getWeight)  
 .thenComparing(Melon::getType);  
  
// Hemi(1600g), Horned(1600g), Gac(2000g), Apollo(3000g), Gac(3000g)  
List<Melon> sortedMelons = melons.stream()  
 .sorted(byWeightAndType)  
 .collect(Collectors.toList());Copy

This flavor of thenComparing() takes a Function as an argument. This Function is used to extract the Comparable sort key. The returned Comparator is applied only when the previous Comparator has found two equal objects.

Another flavor of thenComparing() gets a Comparator:

Comparator<Melon> byWeightAndType = Comparator.comparing(Melon::getWeight)  
 .thenComparing(Comparator.comparing(Melon::getType));Copy

Finally, let's consider the following List of Melon:

List<Melon> melons = Arrays.asList(new Melon("Gac", 2000),  
 new Melon("Horned", 1600), new Melon("Apollo", 3000),  
 new Melon("Gac", 3000), new Melon("hemi", 1600));Copy

We intentionally added a mistake to the last Melon. Its type is lowercase this time. If we apply the byWeightAndType comparator, then the output will be as follows:

Horned(1600g), hemi(1600g), ...Copy

Being a lexicographic-order comparator, byWeightAndType will place Horned before hemi. So, it will be useful to sort by type in a case-insensitive manner. An elegant solution to this problem will rely on another flavor of thenComparing() , which allows us to pass a Function and Comparator as arguments. The Function that is passed extracts the Comparable sort key, and the given Comparator is used to compare this sort key:

Comparator<Melon> byWeightAndType = Comparator.comparing(Melon::getWeight)  
 .thenComparing(Melon::getType, String.CASE\_INSENSITIVE\_ORDER);Copy

This time, the result will be as follows (we are back on track):

hemi(1600g), Horned(1600g),...Copy

For int, long, and double, we have comparingInt(), comparingLong(), comparingDouble(), thenComparingInt(), thenComparingLong(), and thenComparingDouble(). The comparing() and thenComparing() methods come with the same flavors.

Composing functions

Lambda expressions that are represented via the Function interface can be composed via the Function.andThen() and Function.compose() methods.

andThen​(Function<? super R,​? extends V> after) returns a composed Function that does the following:

* Applies this function to its input
* Applies the after function to the result

Let's take a look at an example of this:

Function<Double, Double> f = x -> x \* 2;  
Function<Double, Double> g = x -> Math.pow(x, 2);  
Function<Double, Double> gf = f.andThen(g);  
double resultgf = gf.apply(4d); // 64.0Copy

In this example, the f function is applied to its input (4). The result of applying f is 8 (f(4) = 4 \* 2). This result is the input of the second function, g. The result of applying g is 64 (g(8) = Math.pow(8, 2)). The following diagram depicts the flow for four inputs – **1**, **2**, **3**, and **4**:

Diagrama

Descripción generada automáticamente

So, this is like g(f(x)). The opposite, f(g(x)), can be shaped using Function.compose(). The returned composed function applies the *before function* to its input, and then applies *this function* to the result:

double resultfg = fg.apply(4d); // 32.0Copy

In this example, the g function is applied to its input (4). The result of applying g is 16 (g(4) = Math.pow(4, 2)). This result is the input of the second function, f. The result of applying f is 32 (f(16) = 16 \* 2). The following diagram depicts the flow for four inputs – **1**, **2**, **3**, and **4**:

Diagrama

Descripción generada automáticamente

Based on the same principles, we can develop an application for editing an article by composing the addIntroduction(), addBody(), and addConclusion() methods. Please take a look at the code that's bundled with this book to see an implementation of this.

We can write other pipelines as well by simply juggling this with the composition process.

198. Default methods

Default methods were added to Java 8. Their main goal is to provide support for interfaces so that they can evolve beyond an abstract contract (contain only abstract methods). This facility is very useful for people that write libraries and want to evolve APIs in a compatible way. Via default methods, an interface can be enriched without disrupting existing implementations.

A default method is implemented directly in the interface and is recognized by the default keyword.

For example, the following interface defines an abstract method called area() and a default method called perimeter():

public interface Polygon {  
  
 public double area();  
  
 default double perimeter(double...segments) {  
 return Arrays.stream(segments)  
 .sum();  
 }  
}Copy

Since the perimeter of all common polygons (for example, squares) is the sum of the edges, we can implement it here. On the other hand, the area formula differs from polygon to polygon, and so a default implementation will not be very useful.

Now, let's define a Square class that implements Polygon. Its goal is to express the area of a square via the perimeter:

public class Square implements Polygon {  
  
 private final double edge;  
  
 public Square(double edge) {  
 this.edge = edge;  
 }  
  
 @Override  
 public double area() {  
 return Math.pow(perimeter(edge, edge, edge, edge) / 4, 2);  
 }  
}Copy

Other polygons (for example, rectangles and triangles) can implement Polygon and express the area based on the perimeter that's computed via the default implementation.

However, in certain cases, we may need to override the default implementation of a default method. For example, the Square class may override the perimeter() method, as follows:

@Override  
public double perimeter(double...segments) {  
 return segments[0] \* 4;  
}Copy

We can call it as follows:

@Override  
public double area() {  
 return Math.pow(perimeter(edge) / 4, 2);  
}Copy

Summary

Our job's done! This chapter covered infinite streams, null-safe streams, and default methods. A comprehensive list of problems covered grouping, partitioning, and collectors, including the JDK 12 teeing() collector and writing a custom collector. In addition, takeWhile(), dropWhile(), composing functions, predicates and comparators, testing and debugging lambdas, and other cool topics were covered as well.

Download the applications from this chapter to view the results and additional details.

Concurrency - Thread Pools, Callables, and Synchronizers

This chapter includes 14 problems that involve Java concurrency. We will start with several fundamental problems about thread life cycles and object- and class-level locking. We then continue with a bunch of problems about thread pools in Java including the JDK 8 work-stealing thread pool. After that, we have problems dedicated to Callable and Future. Then, we dedicate several problems to Java synchronizers (for example, barrier, semaphore, and exchanger). By the end of this chapter, you should be familiar with the main coordinates of Java concurrency and be ready to continue with a set of advanced problems.

Problems

Use the following problems to test your concurrency programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Thread life cycle states**: Write several programs that capture each life cycle state of a thread.
2. **Object- versus class-level locking**: Write several examples that exemplify object- versus class-level locking via thread synchronization.
3. **Thread pools in Java**: Provide a brief overview of thread pools in Java.
4. **Thread pool with a single thread**: Write a program that simulates an assembly line for checking and packing up bulbs using two workers.
5. **Thread pool with a fixed number of threads**: Write a program that simulates an assembly line for checking and packing up bulbs using multiple workers.
6. **Cached** **and scheduled thread pools**: Write a program that simulates an assembly line for checking and packing up bulbs using workers as needed (for example, adapt the number of packers (increase or decrease) to ingest the incoming flux produced by the checker).
7. **Work-stealing thread pool**: Write a program that relies on a work-stealing thread pool. More precisely, write a program that simulates an assembly line for checking and packing up bulbs as follows: checking takes place during the day, and packing takes place at night. The checking process results in a queue of 15 million bulbs every day.
8. **Callable and Future**: Write a program that simulates an assembly line for checking and packing up bulbs using Callable and Future.
9. **Invoking multiple Callable tasks**: Write a program that simulates an assembly line for checking and packing up bulbs as follows: checking takes place during the day, and packing takes place at night. The checking process results in a queue of 100 bulbs every day. The packing process should pack and return all the bulbs at once. In other words, we should submit all Callable tasks and wait for all of them to complete.
10. **Latches**: Write a program that relies on CountDownLatch to simulate the process of starting a server. The server is considered started after its internal services have started. Services can be started concurrently and are independent of each other.
11. **Barriers**: Write a program that relies on CyclicBarrier to simulate the process of starting a server. The server is considered started after its internal services have started. Services can be prepared for start concurrently (this is time-consuming), but they run interdependently – therefore, once they are ready to start, they must be started all at once.
12. **Exchangers**: Write a program that simulates using Exchanger, an assembly line for checking and packing up bulbs using two workers. A worker (the checker) is checking bulbs and adding them in a basket. When the basket is full, the worker gives it to the other worker (the packer) from whom they receive an empty basket. The process repeats until the assembly line stops.
13. **Semaphores**: Write a program that simulates using one Semaphore per day at the barbershop. Mainly, our barbershop can serve a maximum of three people at a time (it has only three seats). When a person arrives at the barbershop, they try to take a seat. After they are served by a barber, the person releases the seat. If a person arrives at the barbershop when all three seats are taken, they must wait for a certain amount of time. If this time elapses and no seats have been freed, they will leave the barbershop.
14. **Phasers**: Write a program that relies on Phaser to simulate the process of starting a server in three phases. The server is considered started after its five internal services have started. At the first phase, we need to concurrently start three services. At the second phase, we need to concurrently start two more two services (these can be started only if the first three are already running). At phase three, the server performs a final check-in and is considered started.

Solutions

The following sections describe solutions to the preceding problems. Remember that there usually isn't just one correct way to solve a particular problem. Also, remember that the explanations shown here include only the most interesting and important details needed to solve the problems. Download the example solutions to see additional details and to experiment with the programs at <https://github.com/PacktPublishing/Java-Coding-Problems>.

199. Thread life cycle states

The states of a Java thread are expressed via the Thread.State enumeration. The possible states of a Java thread are shown in the following diagram:

Diagrama

Descripción generada automáticamente

The different lifecycle states are as follows:

* The **NEW** state
* The**RUNNABLE** state
* The **BLOCKED** state
* The**WAITING** state
* The**TIMED\_WAITING** state
* The**TERMINATED** state

Let's learn about all the different states in the following sections.

The NEW state

A Java thread is in the **NEW** state if it is created but not started (the thread constructor creates threads in the **NEW** state). This is its state until the start() method is invoked. The code bundled with this book contains several snippets of code that reveal this state via different construction techniques, including lambdas. For brevity, the following is just one of these constructions:

public class NewThread {  
  
 public void newThread() {  
 Thread t = new Thread(() -> {});  
 System.out.println("NewThread: " + t.getState()); // NEW  
 }  
}  
  
NewThread nt = new NewThread();  
nt.newThread();Copy

The RUNNABLE state

The transition from **NEW** to **RUNNABLE** is obtained by calling the start() method. In this state, a thread can be running or ready to run. When it is ready to run, a thread is waiting for the JVM thread-scheduler to allocate the needed resources and time to run to it. As soon as the processor is available, the thread-scheduler will run the thread.

The following snippet of code should print **RUNNABLE**, since we print the state of the thread after calling start(). But because of thread-scheduler internal mechanisms, this is not guaranteed:

public class RunnableThread {  
  
 public void runnableThread() {  
 Thread t = new Thread(() -> {});  
 t.start();  
  
 // RUNNABLE  
 System.out.println("RunnableThread : " + t.getState());   
 }  
}  
  
RunnableThread rt = new RunnableThread();  
rt.runnableThread();Copy

The BLOCKED state

When a thread is trying to execute I/O tasks or synchronized blocks, it may enter into the **BLOCKED** state. For example, if a thread, t1, tries to enter into a synchronized block of code that is already being accessed by another thread, t2, then t1 is kept in the **BLOCKED** state until it can acquire the lock.

This scenario is shaped in the following snippet of code:

1. Create two threads: t1 and t2.
2. Start t1 via the start() method:
   1. t1 will execute the run() method and will acquire the lock for the synchronized method, syncMethod().
   2. The syncMethod() will keep t1 inside forever, since it has an infinite loop.
3. After two seconds (arbitrary time), start t2 via the start() method:
   1. t2 will execute the run() code and will end up in the **BLOCKED** state since it cannot acquire the lock of syncMethod().

The code snippet is as follows:

public class BlockedThread {  
  
 public void blockedThread() {  
  
 Thread t1 = new Thread(new SyncCode());  
 Thread t2 = new Thread(new SyncCode());  
  
 t1.start();  
 Thread.sleep(2000);  
 t2.start();  
 Thread.sleep(2000);  
  
 System.out.println("BlockedThread t1: "   
 + t1.getState() + "(" + t1.getName() + ")");  
 System.out.println("BlockedThread t2: "   
 + t2.getState() + "(" + t2.getName() + ")");  
  
 System.exit(0);  
 }  
  
 private static class SyncCode implements Runnable {  
  
 @Override  
 public void run() {  
 System.out.println("Thread " + Thread.currentThread().getName()   
 + " is in run() method");  
 syncMethod();  
 }  
  
 public static synchronized void syncMethod() {  
 System.out.println("Thread " + Thread.currentThread().getName()   
 + " is in syncMethod() method");  
  
 while (true) {  
 // t1 will stay here forever, therefore t2 is blocked  
 }  
 }  
 }  
}  
  
BlockedThread bt = new BlockedThread();  
bt.blockedThread();Copy

Here is a possible output (the names of threads may differ from here):

Thread Thread-0 is in run() method  
Thread Thread-0 is in syncMethod() method  
Thread Thread-1 is in run() method  
BlockedThread t1: RUNNABLE(Thread-0)  
BlockedThread t2: BLOCKED(Thread-1)Copy

The WAITING state

A thread, t1, that waits (without a timeout period) for another thread, t2, to finish is in the **WAITING** state.

This scenario is shaped in the following snippet of code:

1. Create a thread: t1.
2. Start t1 via the start() method.
3. In the run() method of t1:
   1. Create another thread: t2.
   2. Start t2 via the start() method.
   3. While t2 is running, call t2.join()—since t2 needs to join t1 (or, in other words, t1 needs to wait for t2 to die), t1 is in the **WAITING** state.
4. In the run() method of t2, t2 prints the state of t1, which should be **WAITING** (while printing the t1 state, t2 is running, therefore t1 is waiting).

The code snippet is as follows:

public class WaitingThread {  
  
 public void waitingThread() {  
 new Thread(() -> {  
 Thread t1 = Thread.currentThread();  
 Thread t2 = new Thread(() -> {  
  
 Thread.sleep(2000);  
 System.out.println("WaitingThread t1: "   
 + t1.getState()); // WAITING  
 });  
  
 t2.start();  
  
 t2.join();  
  
 }).start();  
 }  
}  
  
WaitingThread wt = new WaitingThread();  
wt.waitingThread();Copy

The TIMED\_WAITING state

A thread, t1, that waits for an explicit period of time for another thread, t2, to finish is in the **TIMED\_WAITING** state.

This scenario is shaped in the following snippet of code:

1. Create a thread: t1.
2. Start t1 via the start() method.
3. In the run() method of t1, add a sleep time of two seconds (arbitrary time).
4. While t1 is running, the main thread prints the t1 state—the state should be **TIMED\_WAITING** since t1 is in a sleep() that will expire after two seconds.

The code snippet is as follows:

public class TimedWaitingThread {  
  
 public void timedWaitingThread() {  
 Thread t = new Thread(() -> {  
 Thread.sleep(2000);  
 });  
  
 t.start();  
  
 Thread.sleep(500);  
  
 System.out.println("TimedWaitingThread t: "   
 + t.getState()); // TIMED\_WAITING  
 }  
}  
  
TimedWaitingThread twt = new TimedWaitingThread();  
twt.timedWaitingThread();Copy

The TERMINATED state

A thread that successfully finishes its job or is abnormally interrupted is in the **TERMINATE** state. This is very simple to simulate, as in the following snippet of code (the main thread of the application prints the state of the thread, t—when this is happening, the thread, t, has done its job):

public class TerminatedThread {  
  
 public void terminatedThread() {  
 Thread t = new Thread(() -> {});  
 t.start();  
  
 Thread.sleep(1000);  
  
 System.out.println("TerminatedThread t: "   
 + t.getState()); // TERMINATED  
 }  
}  
  
TerminatedThread tt = new TerminatedThread();  
tt.terminatedThread();Copy

In order to write thread-safe classes, we can consider the following techniques:

* Have no *state* (classes with no instance and static variables)
* Have *state,* but don't share it (for example, use instance variables via Runnable, ThreadLocal, and so on)
* Have *state,* but an immutable *state*
* Use message-passing (for example, as Akka framework)
* Use synchronized blocks
* Use volatile variables
* Use data structures from the java.util.concurrent package
* Use synchronizers (for example, CountDownLatch and Barrier)
* Use locks from the java.util.concurrent.locks package

200. Object- versus class-level locking

In Java, a block of code marked as synchronized can be executed by a single thread at a time. Since Java is a multi-threaded environment (it supports concurrency), it needs a synchronization mechanism to avoid issues specific to concurrent environments (for example, deadlocks and memory consistency).

A thread can achieve locks at the object level or at the class level.

Locking at the object level

Locking at object level can be achieved by marking a non-static block of code or non-static method (the lock object for that method's object) with synchronized. In the following examples, only one thread at a time will be allowed to execute the synchronized method/block on the given instance of the class:

* Synchronized method case:

public class ClassOll {  
 public synchronized void methodOll() {  
 ...  
 }  
}Copy

* Synchronized block of code:

public class ClassOll {  
 public void methodOll() {  
 synchronized(this) {  
 ...  
 }  
 }  
}Copy

* Another synchronized block of code:

public class ClassOll {  
  
 private final Object ollLock = new Object();  
 public void methodOll() {  
 synchronized(ollLock) {  
 ...  
 }  
 }  
}Copy

Lock at the class level

In order to protect static data, locking at the class level can be achieved by marking a static method/block or acquiring a lock on the .class reference with synchronized. In the following examples, only one thread of one of the available instances at runtime will be allowed to execute the synchronized block at a time:

* synchronized static method:

public class ClassCll {  
  
 public synchronized static void methodCll() {  
 ...  
 }  
}Copy

* Synchronized block and lock on .class:

public class ClassCll {  
  
 public void method() {  
 synchronized(ClassCll.class) {  
 ...  
 }  
 }  
}Copy

* Synchronized block of code and lock on some other static object:

public class ClassCll {  
  
 private final static Object aLock = new Object();  
  
 public void method() {  
 synchronized(aLock) {  
 ...  
 }  
 }  
}Copy

Good to know

Here are some common cases that imply synchronizations:

* Two threads can execute concurrently a synchronized static method and a non-static method of the same class (see the OllAndCll class of the P200\_ObjectVsClassLevelLocking app). This works because the threads acquire locks on different objects.
* Two threads cannot concurrently execute two different synchronized static methods (or the same synchronized static method) of the same class (check the TwoCll class of the P200\_ObjectVsClassLevelLocking application). This does not work because the first thread acquires a class-level lock. The following combinations will output, staticMethod1(): Thread-0, therefore, only one static synchronized method is executed by only one thread:

TwoCll instance1 = new TwoCll();  
TwoCll instance2 = new TwoCll();Copy

* Two threads, two instances:

new Thread(() -> {  
 instance1.staticMethod1();  
}).start();  
  
new Thread(() -> {  
 instance2.staticMethod2();  
}).start();Copy

* Two threads, one instance:

new Thread(() -> {  
 instance1.staticMethod1();  
}).start();  
  
new Thread(() -> {  
 instance1.staticMethod2();  
}).start();Copy

* Two threads can concurrently execute non-synchronized, synchronized static, and synchronized non-static methods (check the OllCllAndNoLock class of the P200\_ObjectVsClassLevelLocking application).
* It is safe to call a synchronized method from another synchronized method of the same class that requires the same lock. This works because synchronized is *re-entrant* (as long as it is the same lock, the lock acquired for the first method is used in the second method as well). Check the TwoSyncs class of the P200\_ObjectVsClassLevelLocking application.

As a rule of thumb, the synchronized keyword can be used only with static/non-static methods (not constructors)/code blocks. Avoid synchronizing non-final fields and String literals (instances of String created via new are OK).

201. Thread pools in Java

A thread pool is a collection of threads that can be used to execute tasks. A thread pool is responsible for managing the creation, allocation, and life cycles of its threads and contributing to better performance. Now, let's talk about executors.

Executor

In the java.util.concurrent package, there are a bunch of interfaces dedicated to executing tasks. The simplest one is named Executor. This interface exposes a single method named execute​(Runnable command). Here is an example of executing a single task using this method:

public class SimpleExecutor implements Executor {  
  
 @Override  
 public void execute(Runnable r) {  
 (new Thread(r)).start();  
 }  
}  
  
SimpleExecutor se = new SimpleExecutor();  
  
se.execute(() -> {  
 System.out.println("Simple task executed via Executor interface");  
});Copy

ExecutorService

A more complex and comprehensive interface that provides many additional methods is ExecutorService. This is an enriched version of Executor. Java comes with a fully-fledged implementation of ExecutorService, named ThreadPoolExecutor. This is a thread pool that can be instantiated with a bunch of arguments, as follows:

ThreadPoolExecutor​(  
 int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler)Copy

Here is a short description of each of the arguments instantiated in the preceding code:

* corePoolSize: The number of threads to keep in the pool, even if they are idle (unless allowCoreThreadTimeOut is set)
* maximumPoolSize: The maximum number of allowed threads
* keepAliveTime: When this time has elapsed, the idle threads will be removed from the pool (these are idle threads that exceed corePoolSize)
* unit: The time unit for the keepAliveTime argument
* workQueue: A queue for holding the instances of Runnable (only the Runnable tasks submitted by the execute() method) before they are executed
* threadFactory: This factory is used when the executor creates a new thread
* handler: When ThreadPoolExecutor cannot execute a Runnable due to saturation, this is when the thread bounds and queue capacities are full (for example, workQueue has a fixed size and maximumPoolSize is set as well)—it gives the control and decision to this handler

In order to optimize the pool size, we need to collect the following information:

* Number of CPUs (Runtime.getRuntime().availableProcessors())
* Target CPU utilization (in range, [0, 1])
* Wait time (W)
* Compute time (C)

The following formula helps us to determine the optimal size of the pool:

Number of threads   
 = Number of CPUs \* Target CPU utilization \* (1 + W/C)Copy

As a rule of thumb, for compute-intensive tasks (usually small tasks), it can be a good idea to benchmark the thread pool with the number of threads equal with to number of processors or number of processors + 1 (to prevent potential pauses). For time-consuming and blocking tasks (for example, I/O), a larger pool is better since threads will not be available for scheduling at a high rate. Also, pay attention to interferences with other pools (for example, database connections pools, and socket connection pools).

Let's see an example of ThreadPoolExecutor:

public class SimpleThreadPoolExecutor implements Runnable {  
  
 private final int taskId;  
  
 public SimpleThreadPoolExecutor(int taskId) {  
 this.taskId = taskId;  
 }  
  
 @Override  
 public void run() {  
 Thread.sleep(2000);  
 System.out.println("Executing task " + taskId   
 + " via " + Thread.currentThread().getName());  
 }  
  
 public static void main(String[] args) {  
  
 BlockingQueue<Runnable> queue = new LinkedBlockingQueue<>(5);  
 final AtomicInteger counter = new AtomicInteger();  
  
 ThreadFactory threadFactory = (Runnable r) -> {  
 System.out.println("Creating a new Cool-Thread-"   
 + counter.incrementAndGet());  
  
 return new Thread(r, "Cool-Thread-" + counter.get());  
 };  
  
 RejectedExecutionHandler rejectedHandler  
 = (Runnable r, ThreadPoolExecutor executor) -> {  
 if (r instanceof SimpleThreadPoolExecutor) {  
 SimpleThreadPoolExecutor task=(SimpleThreadPoolExecutor) r;  
 System.out.println("Rejecting task " + task.taskId);  
 }  
 };  
  
 ThreadPoolExecutor executor = new ThreadPoolExecutor(10, 20, 1,  
 TimeUnit.SECONDS, queue, threadFactory, rejectedHandler);  
  
 for (int i = 0; i < 50; i++) {  
 executor.execute(new SimpleThreadPoolExecutor(i));  
 }  
  
 executor.shutdown();  
 executor.awaitTermination(  
 Integer.MAX\_VALUE, TimeUnit.MILLISECONDS);  
 }  
}Copy

The main() method fires 50 instances of Runnable. Each Runnable sleeps for two seconds and prints a message. The work queue is limited to five instances of Runnable—the core threads to 10, the maximum number of threads to 20, and the idle timeout to one second. A possible output will look as follows:

Creating a new Cool-Thread-1  
...  
Creating a new Cool-Thread-20  
Rejecting task 25  
...  
Rejecting task 49  
Executing task 22 via Cool-Thread-18  
...  
Executing task 12 via Cool-Thread-2Copy

ScheduledExecutorService

ScheduledExecutorService is an ExecutorService that can schedule tasks for execution after a given delay, or execute periodically. Here, we have methods such as schedule(), scheduleAtFixedRate(), and scheduleWithFixedDelay​(). While schedule() is used for one-shot tasks, scheduleAtFixedRate() and scheduleWithFixedDelay() are used for periodic tasks.

Thread pools via Executors

One step further, and we introduce the helper class, Executors. This class exposes several types of thread pools using the following methods:

* newSingleThreadExecutor(): This is a thread pool that manages only one thread with an unbounded queue, which only executes one task at a time:

ExecutorService executor   
 = Executors.newSingleThreadExecutor();Copy

* newCachedThreadPool(): This is a thread pool that creates new threads and removes idle threads (after 60 seconds) as they are needed; the core pool size is 0 and the maximum pool size is Integer.MAX\_VALUE (this thread pool expands when demand increases and contracts when demand decreases):

ExecutorService executor = Executors.newCachedThreadPool();Copy

* newFixedThreadPool(): This is a thread pool with a fixed number of threads and an unbounded queue, which creates the effect of an infinite timeout (the core pool size and the maximum pool size are equal to the specified size):

ExecutorService executor = Executors.newFixedThreadPool(5);Copy

* newWorkStealingThreadPool(): This a thread pool based on a work-stealing algorithm (it acts as a layer over a fork/join framework):

ExecutorService executor = Executors.newWorkStealingPool();Copy

* newScheduledThreadPool(): A thread pool that can schedule commands to run after a given delay, or to execute periodically (we can specify the core pool size):

ScheduledExecutorService executor   
 = Executors.newScheduledThreadPool(5);Copy

202. Thread pool with a single thread

In order to show how a thread pool with a single thread works, let's assume that we want to write a program that simulates an assembly line (or a conveyor) for checking and packing up bulbs using two workers.

By *checking*, we understand that the worker tests if the bulb lights up or not. By *packing*, we understand that the worker takes the verified bulb and put it in a box. This kind of process is very common in almost any factory.

The two workers are as follows:

* A so-called producer (or checker) that is responsible for testing each bulb to see if the bulb lights up or not
* A so-called consumer (or packer) that is responsible for packing each checked bulb into a box

This kind of problem is a perfect fit for the producer-consumer design pattern shown in the following diagram:

Diagrama

Descripción generada automáticamente

Most commonly, in this pattern, the producer and consumer communicate via a queue (the producer enqueues data, and the consumer dequeues data). This queue is known as the *data buffer*. Of course, depending on the process design, other data structures can play the role of data buffer as well.

Now, let's see how we can implement this pattern if the producer waits for the consumer to be available.

Later on, we will implement this pattern for a producer that doesn't wait for a consumer.

Producer waits for the consumer to be available

When the assembly line starts, the producer will check the incoming bulbs one by one, while the consumer will pack them (one bulb into each box). This flow repeats until the assembly line stops.

The following diagram is a graphical representation of this flow between the **producer** and the **consumer**:

Diagrama

Descripción generada automáticamente

We can consider the assembly line a helper of our factory, therefore it can be implemented as a helper or utility class (of course, it can be easily switched to a non-static implementation as well, so feel free to do the switch if it makes more sense for your cases):

public final class AssemblyLine {  
  
 private AssemblyLine() {  
 throw new AssertionError("There is a single assembly line!");  
 }  
 ...  
}Copy

Of course, there are many ways to implement this scenario, but we are interested in using the Java ExecutorService, more precisely Executors.newSingleThreadExecutor(). An Executor that uses a single worker thread operating off of an unbounded queue is created by this method.

We have only two workers, so we can use two instances of Executor (an Executor will power up the producer, and another one will power up the consumer). So, the producer will be a thread, and the consumer will be another thread:

private static ExecutorService producerService;  
private static ExecutorService consumerService;Copy

Since the producer and the consumer are good friends, they decide to work based on a simple scenario:

* The producer will check a bulb and pass it to the consumer only if the consumer is not busy (if the consumer is busy, the producer will wait a while until the consumer is free)
* The producer will not check the next bulb until they manage to pass the current bulb to the consumer
* The consumer will pack each incoming bulb as soon as possible

This scenario works well for TransferQueue or SynchronousQueue, which carries out a process very similar to the aforementioned scenario. Let's use TransferQueue. This is a BlockingQueue in which the producers may wait for the consumers to receive elements. BlockingQueue implementations are thread-safe:

private static final TransferQueue<String> queue   
 = new LinkedTransferQueue<>();Copy

The workflow between producer and consumer is of the **First In First Out** type (**FIFO**: the first bulb checked is the first bulb packed) therefore LinkedTransferQueue can be a good choice.

Once the assembly line starts, the producer will continuously check bulbs, therefore we can implement it as a class as follows:

private static final int MAX\_PROD\_TIME\_MS = 5 \* 1000;  
private static final int MAX\_CONS\_TIME\_MS = 7 \* 1000;  
private static final int TIMEOUT\_MS = MAX\_CONS\_TIME\_MS + 1000;  
private static final Random rnd = new Random();  
private static volatile boolean runningProducer;  
...  
private static class Producer implements Runnable {  
  
 @Override  
 public void run() {  
 while (runningProducer) {  
 try {  
 String bulb = "bulb-" + rnd.nextInt(1000);  
  
 Thread.sleep(rnd.nextInt(MAX\_PROD\_TIME\_MS));  
  
 boolean transfered = queue.tryTransfer(bulb,  
 TIMEOUT\_MS, TimeUnit.MILLISECONDS);  
  
 if (transfered) {  
 logger.info(() -> "Checked: " + bulb);  
 }  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 break;  
 }  
 }  
 }  
}Copy

So, the producer transfers a checked bulb to the consumer via the tryTransfer() method. If it is possible to transfer the elements to a consumer before the timeout elapses, this method will do so.

Avoid using the transfer() method, which may block the thread indefinitely.

In order to simulate the time spent by the producer checking a bulb, the corresponding thread will sleep a random number of seconds between 0 and 5 (5 seconds is the maximum time needed to check a bulb). If the consumer is not available after this time, more time will be spent (in tryTransfer()) until the consumer is available or the timeout elapses.

On the other hand, the consumer is implemented using another class, as follows:

private static volatile boolean runningConsumer;  
...  
private static class Consumer implements Runnable {  
  
 @Override  
 public void run() {  
 while (runningConsumer) {  
 try {  
 String bulb = queue.poll(  
 MAX\_PROD\_TIME\_MS, TimeUnit.MILLISECONDS);  
  
 if (bulb != null) {  
 Thread.sleep(rnd.nextInt(MAX\_CONS\_TIME\_MS));  
 logger.info(() -> "Packed: " + bulb);  
 }  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 break;  
 }  
 }  
 }  
}Copy

The consumer may take a bulb from the producer via the queue.take() method. This method retrieves and removes the head of this queue, waiting, if necessary, until a bulb becomes available. Or it may call the poll() method, in which the head of the queue is retrieved and removed, or if this queue is empty it returns null. But neither of these two is right for us. If the producer is not available, the consumer may remain stuck in the take() method. On the other hand, if the queue is empty (the producer is checking the current bulb right now), the poll() method will be called again and again very quickly, causing a dummy repetition. The solution to this is poll​(long timeout, TimeUnit unit). This method retrieves and removes the head of this queue and waits up to the specified wait time, if required, for a bulb to become available. It will return null only if the queue is empty after the waiting time has elapsed.

In order to simulate the time the consumer spends packing a bulb, the corresponding thread will sleep a random number of seconds between 0 and 7 (7 seconds is the maximum time needed for packing a bulb).

Starting the producer and the consumer is a very simple task accomplished in a method named startAssemblyLine(), as follows:

public static void startAssemblyLine() {  
  
 if (runningProducer || runningConsumer) {  
 logger.info("Assembly line is already running ...");  
 return;  
 }  
  
 logger.info("\n\nStarting assembly line ...");  
 logger.info(() -> "Remaining bulbs from previous run: \n"  
 + queue + "\n\n");  
  
 runningProducer = true;  
 producerService = Executors.newSingleThreadExecutor();  
 producerService.execute(producer);  
  
 runningConsumer = true;  
 consumerService = Executors.newSingleThreadExecutor();  
 consumerService.execute(consumer);  
}Copy

Stopping the assembly line is a delicate process that can be tackled via different scenarios. Mainly, when the assembly line is stopped, the producer should check the current bulb as the last bulb and the consumer must pack it. It is possible that the producer will have to wait for the consumer to pack their current bulb before they can transfer the last bulb; further, the consumer must pack this bulb as well.

In order to follow this scenario, we stop the producer first and the consumer second:

public static void stopAssemblyLine() {  
  
 logger.info("Stopping assembly line ...");  
  
 boolean isProducerDown = shutdownProducer();  
 boolean isConsumerDown = shutdownConsumer();  
  
 if (!isProducerDown || !isConsumerDown) {  
 logger.severe("Something abnormal happened during  
 shutting down the assembling line!");  
  
 System.exit(0);  
 }  
  
 logger.info("Assembling line was successfully stopped!");  
}  
  
private static boolean shutdownProducer() {  
 runningProducer = false;  
 return shutdownExecutor(producerService);  
}  
  
private static boolean shutdownConsumer() {  
 runningConsumer = false;  
 return shutdownExecutor(consumerService);  
}Copy

Finally, we give enough time to the producer and consumer to stop normally (without the interruption of threads). This takes place in the shutdownExecutor() method, as follows:

private static boolean shutdownExecutor(ExecutorService executor) {  
  
 executor.shutdown();  
  
 try {  
 if (!executor.awaitTermination(TIMEOUT\_MS \* 2,  
 TimeUnit.MILLISECONDS)) {  
 executor.shutdownNow();  
 return executor.awaitTermination(TIMEOUT\_MS \* 2,  
 TimeUnit.MILLISECONDS);  
 }  
  
 return true;  
 } catch (InterruptedException ex) {  
 executor.shutdownNow();  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 }  
  
 return false;  
}Copy

The first thing that we do is set the runningProducer static variable to false. This will break while(runningProducer), therefore this will be the last bulb checked. Further, we initiate the shutdown procedure for the producer.

In the case of a consumer, the first thing that we do is set the runningConsumer static variable to false. This will break while(runningConsumer), therefore this will be the last bulb packed. Further, we initiate the shutdown procedure for the consumer.

Let's see a possible execution of the assembly line (run it for 10 seconds):

AssemblyLine.startAssemblyLine();  
Thread.sleep(10 \* 1000);  
AssemblyLine.stopAssemblyLine();Copy

A possible output will be as follows:

Starting assembly line ...  
...  
[2019-04-14 07:39:40] [INFO] Checked: bulb-89  
[2019-04-14 07:39:43] [INFO] Packed: bulb-89  
...  
Stopping assembly line ...  
...  
[2019-04-14 07:39:53] [INFO] Packed: bulb-322  
Assembling line was successfully stopped!Copy

Generally speaking, if it takes a lot of time to stop the assembly line (it acts as if it were blocked), then there's probably an unbalanced rate between the number of producers and consumers and/or between the production and consumption times. You may need to add or subtract producers or consumers.

Producer doesn't wait for the consumer to be available

If the producer can check bulbs faster than the consumer can pack them, then most probably they will decide to have the following workflow:

* The producer will check bulbs one by one and push them in a queue
* The consumer will poll from the queue and pack the bulbs

Since the consumer is slower than the producer, the queue will hold checked but unpacked bulbs (we may assume that there is a low chance to have an empty queue). In the following diagram, we have the producer, the consumer, and the queue used for storing checked but unpacked bulbs:

Diagrama

Descripción generada automáticamente

In order to shape this scenario, we can rely on ConcurrentLinkedQueue (or LinkedBlockingQueue). This is an unbounded thread-safe queue based on linked nodes:

private static final Queue<String> queue   
 = new ConcurrentLinkedQueue<>();Copy

In order to push a bulb in the queue, the producer calls the offer() method:

queue.offer(bulb);Copy

On the other hand, the consumer processes bulbs from the queue using the poll() method (since the consumer is slower than the producer, it should be a rare case when poll() will return null):

String bulb = queue.poll();Copy

Let's start the assembly line for the first time for 10 seconds. This will output the following:

Starting assembly line ...  
...  
[2019-04-14 07:44:58] [INFO] Checked: bulb-827  
[2019-04-14 07:44:59] [INFO] Checked: bulb-257  
[2019-04-14 07:44:59] [INFO] Packed: bulb-827  
...  
Stopping assembly line ...  
...  
[2019-04-14 07:45:08] [INFO] Checked: bulb-369  
[2019-04-14 07:45:09] [INFO] Packed: bulb-690  
...  
Assembling line was successfully stopped!Copy

At this point, the assembly line is stopped, and in the queue, we have the following (these bulbs have been checked, but not packed):

[**bulb-968**, **bulb-782**, **bulb-627**, bulb-886, ...]Copy

We restart the assembly line and check the highlighted lines, which reveal that the consumer resumes its job from where they'd stopped:

Starting assembly line ...  
**[2019-04-14 07:45:12] [INFO ] Packed: bulb-968**[2019-04-14 07:45:12] [INFO ] Checked: bulb-812  
[2019-04-14 07:45:12] [INFO ] Checked: bulb-470  
**[2019-04-14 07:45:14] [INFO ] Packed: bulb-782**[2019-04-14 07:45:15] [INFO ] Checked: bulb-601  
**[2019-04-14 07:45:16] [INFO ] Packed: bulb-627**...Copy

203. Thread pool with a fixed number of threads

This problem reiterates the scenario from the *Thread pool with a single thread* section. This time, the assembly line uses three producers and two consumers, as in the following diagram:

Diagrama

Descripción generada automáticamente

We can rely on Executors.newFixedThreadPool​(int nThreads) to simulate the fixed number of producers and consumers. We allocate one thread per producer (respectively, consumer), therefore the code is pretty simple:

private static final int PRODUCERS = 3;  
private static final int CONSUMERS = 2;  
private static final Producer producer = new Producer();  
private static final Consumer consumer = new Consumer();  
private static ExecutorService producerService;  
private static ExecutorService consumerService;  
...  
producerService = Executors.newFixedThreadPool(PRODUCERS);  
for (int i = 0; i < PRODUCERS; i++) {  
 producerService.execute(producer);  
}  
  
consumerService = Executors.newFixedThreadPool(CONSUMERS);  
for (int i = 0; i < CONSUMERS; i++) {  
 consumerService.execute(consumer);  
}Copy

The queue in which the producers can add the checked bulbs can be of the LinkedTransferQueue or ConcurrentLinkedQueue type, and so on.

The complete source code based on LinkedTransferQueue and ConcurrentLinkedQueue can be found in the code bundled with this book.

204. Cached and scheduled thread pools

This problem reiterates the scenario from the *Thread pool with a single thread* section. This time, we assume that the producer (more than one producer can be used as well) checks a bulb in no more than one second. Moreover, a consumer (packer) needs a maximum of 10 seconds to pack a bulb. The producer and consumer times can be shaped as follows:

private static final int MAX\_PROD\_TIME\_MS = 1 \* 1000;  
private static final int MAX\_CONS\_TIME\_MS = 10 \* 1000;Copy

Obviously, in these conditions, one consumer cannot face the incoming flux. The queue used for storing bulbs until they are packed will continuously increase. The producer will add to this queue much faster than the consumer can poll. Therefore, more consumers are needed, as in the following diagram:

Diagrama

Descripción generada automáticamente

Since there is a single producer, we can rely on Executors.newSingleThreadExecutor():

private static volatile boolean runningProducer;  
private static ExecutorService producerService;  
private static final Producer producer = new Producer();  
...  
public static void startAssemblyLine() {  
 ...  
 runningProducer = true;  
 producerService = Executors.newSingleThreadExecutor();  
 producerService.execute(producer);  
 ...  
}Copy

The Producer is almost the same as in the previous problems except for the extraProdTime variable:

private static int extraProdTime;  
private static final Random rnd = new Random();  
...  
private static class Producer implements Runnable {  
  
 @Override  
 public void run() {  
 while (runningProducer) {  
 try {  
 String bulb = "bulb-" + rnd.nextInt(1000);  
 Thread.sleep(rnd.nextInt(MAX\_PROD\_TIME\_MS) + extraProdTime);  
 queue.offer(bulb);  
  
 logger.info(() -> "Checked: " + bulb);  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 break;  
 }  
 }  
 }  
}Copy

The extraProdTime variable is initially 0. This will be needed when we slow down the producer:

Thread.sleep(rnd.nextInt(MAX\_PROD\_TIME\_MS) + extraProdTime);Copy

After running at a high speed for a while, the producer will get tired and will need more time to check each bulb. If the producer slows down the production rate, the number of consumers should be decreased too.

When the producer runs at a high speed, we will need more consumers (packers). But how many? Using a fixed number of consumers (newFixedThreadPool()) will raise at least two drawbacks:

* If the producer slows down at some moment, some consumers will remain without work and will simply stick around
* If the producer becomes even more efficient, more consumers are needed to face the incoming flux

Basically, we should be able to vary the number of consumers depending on producer efficiency.

For these kinds of jobs, we have Executors.newCachedThreadPool​(). A cached thread pool will reuse the existing threads and will create new ones as needed (we can add more consumers). Threads are terminated and removed from the cache if they have not been used for 60 seconds (we can remove consumers).

Let's start with a single active consumer:

private static volatile boolean runningConsumer;  
private static final AtomicInteger   
 nrOfConsumers = new AtomicInteger();  
private static final ThreadGroup threadGroup   
 = new ThreadGroup("consumers");  
private static final Consumer consumer = new Consumer();  
private static ExecutorService consumerService;  
...  
public static void startAssemblyLine() {  
 ...  
 runningConsumer = true;  
 consumerService = Executors  
 .newCachedThreadPool((Runnable r) -> new Thread(threadGroup, r));  
 nrOfConsumers.incrementAndGet();  
 consumerService.execute(consumer);  
 ...  
}Copy

Because we want to be able to see how many threads (consumers) are active at one moment, we add them in a ThreadGroup via a custom ThreadFactory:

consumerService = Executors  
 .newCachedThreadPool((Runnable r) -> new Thread(threadGroup, r));Copy

Later, we will be able to fetch the number of active consumers using the following code:

threadGroup.activeCount();Copy

Knowing the number of active consumers is a good indicator that can be combined with the current size of the bulb queue for determining whether more consumers are needed.

The consumer implementation is listed as follows:

private static class Consumer implements Runnable {  
  
 @Override  
 public void run() {  
  
 while (runningConsumer && queue.size() > 0  
 || nrOfConsumers.get() == 1) {  
 try {  
 String bulb = queue.poll(MAX\_PROD\_TIME\_MS   
 + extraProdTime, TimeUnit.MILLISECONDS);  
  
 if (bulb != null) {  
 Thread.sleep(rnd.nextInt(MAX\_CONS\_TIME\_MS));  
 logger.info(() -> "Packed: " + bulb + " by consumer: "   
 + Thread.currentThread().getName());  
 }  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 break;  
 }  
 }  
  
 nrOfConsumers.decrementAndGet();  
 logger.warning(() -> "### Thread " +  
 Thread.currentThread().getName()   
 + " is going back to the pool in 60 seconds for now!");  
 }  
}Copy

Assuming that the assembly line is running, a consumer will continue to pack bulbs as long as the queue is not empty or they are the only consumer left (we can't have 0 consumers). We can interpret that an empty queue means too many consumers are there. So, when a consumer sees that the queue is empty and they are not the only working consumer, they become idle (in 60 seconds, they will be automatically removed from the cached thread pool).

Do not confuse nrOfConsumers with threadGroup.activeCount(). The nrOfConsumers variable stores the number of consumers (threads) who pack bulbs right now, while threadGroup.activeCount() represents all active consumers (threads) including those that are not working right now (idle) and are just waiting to be reused or dispatched from the cache.

Now, in a real case, a supervisor will monitor the assembly line and when they notice that the current number of consumers cannot face the incoming influx, they will call more consumers to join (a maximum of 50 consumers are allowed). Moreover, when they notice that some consumers are just sticking around, they will dispatch them to other jobs. The following diagram is a graphical representation of this scenario:

Diagrama

Descripción generada automáticamente

For testing purposes, our supervisor, newSingleThreadScheduledExecutor(), will be a single-threaded executor that can schedule the given commands to run after a specified delay. It may also execute the commands periodically:

private static final int MAX\_NUMBER\_OF\_CONSUMERS = 50;  
private static final int MAX\_QUEUE\_SIZE\_ALLOWED = 5;  
private static final int MONITOR\_QUEUE\_INITIAL\_DELAY\_MS = 5000;  
private static final int MONITOR\_QUEUE\_RATE\_MS = 3000;  
private static ScheduledExecutorService monitorService;  
...  
private static void monitorQueueSize() {  
  
 monitorService = Executors.newSingleThreadScheduledExecutor();  
  
 monitorService.scheduleAtFixedRate(() -> {  
 if (queue.size() > MAX\_QUEUE\_SIZE\_ALLOWED   
 && threadGroup.activeCount() < MAX\_NUMBER\_OF\_CONSUMERS) {  
 logger.warning("### Adding a new consumer (command) ...");  
  
 nrOfConsumers.incrementAndGet();  
 consumerService.execute(consumer);  
 }  
  
 logger.warning(() -> "### Bulbs in queue: " + queue.size()   
 + " | Active threads: " + threadGroup.activeCount()   
 + " | Consumers: " + nrOfConsumers.get()   
 + " | Idle: " + (threadGroup.activeCount()   
 - nrOfConsumers.get()));  
 }, MONITOR\_QUEUE\_INITIAL\_DELAY\_MS, MONITOR\_QUEUE\_RATE\_MS,  
 TimeUnit.MILLISECONDS);  
}Copy

We rely on scheduleAtFixedRate() to monitor the assembly line every three seconds with an initial delay of five seconds. So, in every three seconds, the supervisor checks the bulb queue size. If there are more than five bulbs in the queue and fewer than 50 consumers, the supervisor requests a new consumer to join the assembly line. If the queue contains five or fewer bulbs or there are already 50 consumers, the supervisor doesn't take any action.

If we start the assembly line now, we can see how the number of consumers increases until the queue size is fewer than six. A possible snapshot will be as follows:

Starting assembly line ...  
[11:53:20] [INFO] Checked: bulb-488  
...  
[11:53:24] [WARNING] ### Adding a new consumer (command) ...  
[11:53:24] [WARNING] ### Bulbs in queue: 7   
 | Active threads: 2   
 | Consumers: 2   
 | Idle: 0  
[11:53:25] [INFO] Checked: bulb-738  
...  
[11:53:36] [WARNING] ### Bulbs in queue: 23   
 | Active threads: 6  
 | Consumers: 6  
 | Idle: 0  
...Copy

When there are more threads than needed, some of them become idle. If for 60 seconds they don't receive a job, they are removed from the cache. If a job occurs when there is no idle thread, a new thread will be created. This process is repeated constantly until we notice a balance in the assembly line. After a while, things start to calm down and the proper number of consumers will be in a small range (small fluctuations). This happens because the producer outputs at a random speed bounded up by a maximum of one second.

After a while (for example, after 20 seconds), let's slow down the producer by four seconds (so, a bulb can be checked in a maximum of five seconds now):

private static final int SLOW\_DOWN\_PRODUCER\_MS = 20 \* 1000;  
private static final int EXTRA\_TIME\_MS = 4 \* 1000;Copy

This can be done using another newSingleThreadScheduledExecutor(), as follows:

private static void slowdownProducer() {  
  
 slowdownerService = Executors.newSingleThreadScheduledExecutor();  
  
 slowdownerService.schedule(() -> {  
 logger.warning("### Slow down producer ...");  
 extraProdTime = EXTRA\_TIME\_MS;  
 }, SLOW\_DOWN\_PRODUCER\_MS, TimeUnit.MILLISECONDS);  
}Copy

This will happen only once, 20 seconds after starting the assembly line. Since the producer speed was decreased by four seconds, there is no need to have the same number of consumers to maintain a queue maximum of five bulbs.

This is revealed in the output, as shown (notice that, at some moments, there is only one consumer needed to handle the queue):

...  
[11:53:36] [WARNING] ### Bulbs in queue: 23   
 | Active threads: 6  
 | Consumers: 6  
 | Idle: 0  
...  
[11:53:39] [WARNING] ### Slow down producer ...  
...  
[11:53:56] [WARNING] ### Thread Thread-5 is going  
 back to the pool in 60 seconds for now!  
[11:53:56] [INFO] Packed: bulb-346 by consumer: Thread-2  
...  
[11:54:36] [WARNING] ### Bulbs in queue: 1   
 | Active threads: 12  
 | Consumers: 1  
 | Idle: 11  
...  
[11:55:48] [WARNING] ### Bulbs in queue: 3   
 | Active threads: 1  
 | Consumers: 1   
 | Idle: 0  
...  
Assembling line was successfully stopped!Copy

Starting the supervisor takes place after starting the assembly line:

public static void startAssemblyLine() {  
 ...  
 monitorQueueSize();  
 slowdownProducer();  
}Copy

The complete application is available in the code bundled with this book.

When using cached-thread pools, pay attention to the number of threads created to accommodate the number of submitted tasks. While for single-thread and fixed-thread pools, we control the number of created threads, a cached-pool can decide to create too many threads. Basically, creating threads uncontrollably may run out of resources quickly. So, in systems that are vulnerable to overload, it's better to rely on fixed-thread pools.

205. Work-stealing thread pool

Let's focus on the packing process, which should be implemented via a work-stealing thread pool. To start, let's discuss what a work-stealing thread pool is, and let's do it via a comparison with a classic thread pool. The following diagram depicts how a classic thread pool works:

Diagrama

Descripción generada automáticamente

So, a thread pool relies on an internal inbound queue to store tasks. Each thread must dequeue a task and execute it. This is suitable for cases when the tasks are time-consuming and their number is relatively low. On the other hand, if these tasks are many and are small (they require a small amount of time to be executed), there will be a lot of contentions as well. This is not good, and even if this is a lock-free queue the problem is not entirely solved.

In order to reduce contentions and increase performance, a thread pool can rely on a work-stealing algorithm and a queue per thread. In this case, there is a central inbound queue for all tasks, and an extra queue (known as the local task queue) for each thread (worker thread), as in the following diagram:

Diagrama

Descripción generada automáticamente

So, each thread will dequeue tasks from the central queue and enqueue them in their own queue. Each thread has its own local queue of tasks. Further, when a thread wants to process a task, it simply dequeues a task from its own local queue. As long as its local queue is not empty, the thread will continue to process the tasks from it without bothering other threads (no contentions with other threads). When its local queue is empty (as in the case of **Thread 2** in the preceding diagram), it tries to steal (via a working-stealing algorithm) tasks from local queues that belong to other threads (for example, **Thread 2** steals tasks from **Thread 3**). If it doesn't find anything to steal, it accesses the shared central inbound queue.

Each local queue is actually a **deque** (short for **double-ended queue**), therefore it can be accessed efficiently from both ends. The thread sees its deque as a stack, meaning that it will enqueue (add new tasks) and dequeue (take tasks for processing) from only one end. On the other hand, when a thread tries to steal from the queue of another thread, it will access the other end (for example, **Thread 2** steals from **Thread 3** queue from the other end). So, tasks are processed from one end and stolen from the other end.

If two threads try to steal from the same local queue then there is contention, but normally this should be insignificant.

What we've just described is the fork/join framework introduced in JDK 7 and exemplified in the *The fork/join framework* section. Starting with JDK 8, the Executors class was enriched with a work-stealing thread pool using the number of available processors as its target parallelism level. This is available via Executors.newWorkStealingPool() and Executors.newWorkStealingPool​(int parallelism).

Let's see the source code of this thread pool:

public static ExecutorService newWorkStealingPool() {  
  
 return new ForkJoinPool(Runtime.getRuntime().availableProcessors(),  
 ForkJoinPool.defaultForkJoinWorkerThreadFactory,  
 null, true);  
}Copy

So, internally, this thread pool instantiates ForkJoinPool via the following constructor:

public ForkJoinPool​(int parallelism,  
 ForkJoinPool.ForkJoinWorkerThreadFactory factory,  
 Thread.UncaughtExceptionHandler handler,  
 boolean asyncMode)Copy

We have the parallelism level set to availableProcessors(), the default thread factory for returning new threads, Thread.UncaughtExceptionHandler, passed as null, and asyncMode set to true. Setting asyncMode to true means that it empowers the local **First In First Out** (**FIFO**) scheduling mode for tasks that are forked and never joined. This mode may be more suitable than the default one (locally stack-based) in programs that rely on worker threads to process only event-style asynchronous tasks.

Nevertheless, don't forget that the local task queue and work-stealing algorithm are empowered only if the worker threads schedule new tasks in their own local queues. Otherwise, ForkJoinPool is just a ThreadPoolExecutor with extra overhead.

When we work directly with ForkJoinPool, we can instruct tasks to explicitly schedule new tasks during execution using ForkJoinTask (typically, via RecursiveTask or RecursiveAction).

But since newWorkStealingPool() is a higher level of abstraction for ForkJoinPool, we cannot instruct tasks to explicitly schedule new tasks during execution. Therefore, newWorkStealingPool() will decide internally how to work based on the tasks that we pass. We can try a comparison between newWorkStealingPool(), newCachedThreadPool(), and newFixedThreadPool(), and see how they perform in two scenarios:

* For a large number of small tasks
* For a small number of time-consuming tasks

Let's take a look at the solutions for both these scenarios in the next sections.

A large number of small tasks

Since the producers (checkers) and consumer (packers) don't work at the same time, we can easily fill up a queue with 15,000,000 bulbs via a trivial for loop (we are not very interested in this part of the assembly line). This is shown in the following code snippet:

private static final Random rnd = new Random();  
private static final int MAX\_PROD\_BULBS = 15\_000\_000;  
private static final BlockingQueue<String> queue   
 = new LinkedBlockingQueue<>();  
...  
private static void simulatingProducers() {  
 logger.info("Simulating the job of the producers overnight ...");  
 logger.info(() -> "The producers checked "   
 + MAX\_PROD\_BULBS + " bulbs ...");  
  
 for (int i = 0; i < MAX\_PROD\_BULBS; i++) {  
 queue.offer("bulb-" + rnd.nextInt(1000));  
 }  
}Copy

Further, let's create a default work-stealing thread pool:

private static ExecutorService consumerService   
 = Executors.newWorkStealingPool();Copy

For comparison, we will also use the following thread pools:

* A cached thread pool:

private static ExecutorService consumerService   
 = Executors.newCachedThreadPool();Copy

* A fixed thread pool using the number of available processors as the number of threads (the number of processors is used by the default work-stealing thread pool as the parallelism level):

private static final Consumer consumer = new Consumer();  
private static final int PROCESSORS   
 = Runtime.getRuntime().availableProcessors();  
private static ExecutorService consumerService   
 = Executors.newFixedThreadPool(PROCESSORS);Copy

And, let's start 15,000,000 small tasks:

for (int i = 0; i < queueSize; i++) {  
 consumerService.execute(consumer);  
}Copy

The Consumer wraps a simple queue.poll() operation, therefore it should run pretty fast, as shown in the following snippet:

private static class Consumer implements Runnable {  
  
 @Override  
 public void run() {  
 String bulb = queue.poll();  
  
 if (bulb != null) {  
 // nothing  
 }  
 }  
}Copy

The following graph represents the collected data for 10 runs:

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Even if this is not a professional benchmark, we can see that the work-stealing thread pool has obtained the best results, while the cached thread poll has the worse results.

A small number of time-consuming tasks

Instead of filling a queue with 15,000,000 bulbs, let's fill 15 queues with 1,000,000 bulbs each:

private static final int MAX\_PROD\_BULBS = 15 \_000\_000;  
private static final int CHUNK\_BULBS = 1 \_000\_000;  
private static final Random rnd = new Random();  
private static final Queue<BlockingQueue<String>> chunks   
 = new LinkedBlockingQueue<>();  
...  
private static Queue<BlockingQueue<String>> simulatingProducers() {  
 logger.info("Simulating the job of the producers overnight ...");  
 logger.info(() -> "The producers checked "   
 + MAX\_PROD\_BULBS + " bulbs ...");  
  
 int counter = 0;  
 while (counter < MAX\_PROD\_BULBS) {  
 BlockingQueue chunk = new LinkedBlockingQueue<>(CHUNK\_BULBS);  
  
 for (int i = 0; i < CHUNK\_BULBS; i++) {  
 chunk.offer("bulb-" + rnd.nextInt(1000));  
 }  
  
 chunks.offer(chunk);  
 counter += CHUNK\_BULBS;  
 }  
  
 return chunks;  
}Copy

And, let's fire up 15 tasks using the following code:

while (!chunks.isEmpty()) {  
 Consumer consumer = new Consumer(chunks.poll());  
 consumerService.execute(consumer);  
}Copy

Each Consumer loops 1,000,000 bulbs using this code:

private static class Consumer implements Runnable {  
  
 private final BlockingQueue<String> bulbs;  
  
 public Consumer(BlockingQueue<String> bulbs) {  
 this.bulbs = bulbs;  
 }  
  
 @Override  
 public void run() {  
 while (!bulbs.isEmpty()) {  
 String bulb = bulbs.poll();  
  
 if (bulb != null) {}  
 }  
 }  
}Copy

The following graph represents the collected data for 10 runs:

Gráfico, Gráfico de líneas

Descripción generada automáticamente

This time, it looks like the work-stealing thread pool worked as a regular thread pool.

206. Callable and Future

This problem reiterates the scenario from the *Thread pool with a single thread* section. We want a single producer and consumer that follow this scenario:

1. An automatic system sends a request to the producer, saying, *check this bulb and if it is ok then return it to me, otherwise tell me what went wrong with this bulb*.
2. The automatic system waits for the producer to check the bulb.
3. When the automatic system receives the checked bulb, it is then passed further to the consumer (packer) and repeats the process.
4. If a bulb has a defect, the producer throws an exception (DefectBulbException) and the automatic system will inspect the cause of the problem.

This scenario is depicted in the following diagram:

Diagrama

Descripción generada automáticamente

In order to shape this scenario, the producer should be able to return a result and throw an exception. Since our producer is a Runnable, it can't do either of these. But Java defines an interface that is named Callable. This is a functional interface with a method named call(). In contrast to the run() method of Runnable, the call() method can return a result and even throw an exception, V call() throws Exception.

This means that the producer (checker) can be written as follows:

private static volatile boolean runningProducer;  
private static final int MAX\_PROD\_TIME\_MS = 5 \* 1000;  
private static final Random rnd = new Random();  
...  
private static class Producer implements Callable {  
  
 private final String bulb;  
  
 private Producer(String bulb) {  
 this.bulb = bulb;  
 }  
  
 @Override  
 public String call()   
 throws DefectBulbException, InterruptedException {  
  
 if (runningProducer) {  
 Thread.sleep(rnd.nextInt(MAX\_PROD\_TIME\_MS));  
  
 if (rnd.nextInt(100) < 5) {  
 throw new DefectBulbException("Defect: " + bulb);  
 } else {  
 logger.info(() -> "Checked: " + bulb);  
 }  
  
 return bulb;  
 }  
  
 return "";  
 }  
}Copy

The executor service can submit a task to a Callable via the submit() method, but it doesn't know when the result of the submitted task will be available. Therefore, Callable immediately returns a special type named, Future. The result of an asynchronous computation is represented by a Future—via Future we can fetch the result of the task when it is available. Conceptually speaking, we can think of a Future as a JavaScript promise, or as a result of a computation that will be done at a later point in time. Now, let's create a Producer and submit it to a Callable:

String bulb = "bulb-" + rnd.nextInt(1000);  
Producer producer = new Producer(bulb);  
  
Future<String> bulbFuture = producerService.submit(producer);  
// this line executes immediatelyCopy

Since the Callable immediately returns a Future, we can perform other tasks while waiting for the result of the submitted task (the isDone() flag method returns true if this task is completed):

while (!future.isDone()) {  
 System.out.println("Do something else ...");  
}Copy

Retrieving the result of Future can be done using the blocking method, Future.get(). This method blocks until the result is available or the specified timeout elapsed (if the result is not available before the timeout, a TimeoutException is thrown):

String checkedBulb = bulbFuture.get(  
 MAX\_PROD\_TIME\_MS + 1000, TimeUnit.MILLISECONDS);  
  
// this line executes only after the result is availableCopy

Once the result is available, we can pass it to Consumer and submit another task to Producer. This cycle repeats as long as the consumer and the producer are running. The code for this is as follows:

private static void automaticSystem() {  
  
 while (runningProducer &amp;&amp; runningConsumer) {  
 String bulb = "bulb-" + rnd.nextInt(1000);  
  
 Producer producer = new Producer(bulb);  
 Future<String> bulbFuture = producerService.submit(producer);  
 ...  
 String checkedBulb = bulbFuture.get(  
 MAX\_PROD\_TIME\_MS + 1000, TimeUnit.MILLISECONDS);  
  
 Consumer consumer = new Consumer(checkedBulb);  
 if (runningConsumer) {  
 consumerService.execute(consumer);  
 }  
 }  
 ...  
}Copy

The Consumer is still a Runnable, therefore it cannot return a result or throw an exception:

private static final int MAX\_CONS\_TIME\_MS = 3 \* 1000;  
...  
private static class Consumer implements Runnable {  
  
 private final String bulb;  
  
 private Consumer(String bulb) {  
 this.bulb = bulb;  
 }  
  
 @Override  
 public void run() {  
 if (runningConsumer) {  
 try {  
 Thread.sleep(rnd.nextInt(MAX\_CONS\_TIME\_MS));  
 logger.info(() -> "Packed: " + bulb);  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 }  
 }  
 }  
}Copy

Finally, we need to start the automatic system. The code for this is as follows:

public static void startAssemblyLine() {  
 ...  
 runningProducer = true;  
 consumerService = Executors.newSingleThreadExecutor();  
  
 runningConsumer = true;  
 producerService = Executors.newSingleThreadExecutor();  
  
 new Thread(() -> {  
 automaticSystem();  
 }).start();  
}Copy

Notice that we don't want to block the main thread, therefore we start the automatic system in a new thread. This way the main thread can control the start-stop process of the assembly line.

Let's run the assembly line for several minutes to collect some output:

Starting assembly line ...  
[08:38:41] [INFO ] Checked: bulb-879  
...  
[08:38:52] [SEVERE ] Exception: DefectBulbException: Defect: bulb-553  
[08:38:53] [INFO ] Packed: bulb-305  
...Copy

OK, the job is done! Let's tackle the final topic here.

Canceling a Future

A Future can be canceled. This is accomplished using the cancel​(boolean mayInterruptIfRunning) method. If we pass it as true, the thread that executes the task is interrupted, otherwise, the thread may complete the task. This method returns true if the task was successfully canceled, otherwise it returns false (typically, because it has already completed normally). Here is a simple example that cancels a task if it takes more than one second to run:

long startTime = System.currentTimeMillis();  
  
Future<String> future = executorService.submit(() -> {  
 Thread.sleep(3000);  
  
 return "Task completed";  
});  
  
while (!future.isDone()) {  
 System.out.println("Task is in progress ...");  
 Thread.sleep(100);  
  
 long elapsedTime = (System.currentTimeMillis() - startTime);  
  
 if (elapsedTime > 1000) {  
 future.cancel(true);  
 }  
}Copy

The isCancelled() method will return true if the task was canceled before it completes normally:

System.out.println("Task was cancelled: " + future.isCancelled()   
 + "\nTask is done: " + future.isDone());Copy

The output will be as follows:

Task is in progress ...  
Task is in progress ...  
...  
Task was cancelled: true  
Task is done: trueCopy

Here are some bonus examples:

* Using Callable and lambdas:

Future<String> future = executorService.submit(() -> {  
 return "Hello to you!";  
});Copy

* Getting a Callable that returns null via Executors.callable​(Runnable task):

Callable<Object> callable = Executors.callable(() -> {  
 System.out.println("Hello to you!");  
});  
  
Future<Object> future = executorService.submit(callable);Copy

* Getting a Callable that returns a result (T) via Executors.callable​(Runnable task, T result):

Callable<String> callable = Executors.callable(() -> {  
 System.out.println("Hello to you!");  
}, "Hi");  
  
Future<String> future = executorService.submit(callable);Copy

207. Invoking multiple Callable tasks

Since the producers (checkers) don't work at the same time with the consumers (packers), we can just simulate their work via a for that adds 100 checked bulbs in a queue:

private static final BlockingQueue<String> queue   
 = new LinkedBlockingQueue<>();  
...  
private static void simulatingProducers() {  
  
 for (int i = 0; i < MAX\_PROD\_BULBS; i++) {  
 queue.offer("bulb-" + rnd.nextInt(1000));  
 }  
}Copy

Now, the consumers must pack each bulb and return it. This means that the Consumer is a Callable:

private static class Consumer implements Callable {  
  
 @Override  
 public String call() throws InterruptedException {  
 String bulb = queue.poll();  
  
 Thread.sleep(100);  
  
 if (bulb != null) {  
 logger.info(() -> "Packed: " + bulb + " by consumer: "   
 + Thread.currentThread().getName());  
  
 return bulb;  
 }  
  
 return "";  
 }  
}Copy

But remember that we should submit all Callable tasks and wait for all of them to complete. This can be achieved via the ExecutorService.invokeAll() method. This method takes a collection of tasks (Collection<? extends Callable<T>>) and returns a list of instances of Future (List<Future<T>>) as an argument. Any call to Future.get() will be blocked until all the instances of Future are complete.

So, first we create a list of 100 tasks:

private static final Consumer consumer = new Consumer();  
...  
List<Callable<String>> tasks = new ArrayList<>();  
for (int i = 0; i < queue.size(); i++) {  
 tasks.add(consumer);  
}Copy

Further, we execute all these tasks and get the list of Future:

private static ExecutorService consumerService  
 = Executors.newWorkStealingPool();  
...  
List<Future<String>> futures = consumerService.invokeAll(tasks);Copy

Finally, we process (in this case, display) the results:

for (Future<String> future: futures) {  
 String bulb = future.get();  
 logger.info(() -> "Future done: " + bulb);  
}Copy

Notice that the first call to the future.get() statement blocks until all the instances of Future are complete. This will lead to the following output:

[12:06:41] [INFO] Packed: bulb-595 by consumer: ForkJoinPool-1-worker-9  
...  
[12:06:42] [INFO] Packed: bulb-478 by consumer: ForkJoinPool-1-worker-15  
[12:06:43] [INFO] Future done: bulb-595  
...Copy

Sometimes, we want to submit several tasks and wait for any one of them to complete. This can be achieved via ExecutorService.invokeAny(). Exactly like invokeAll(), this method gets as an argument a collection of tasks (Collection<? extends Callable<T>>). But it returns the result of the fastest task (not a Future) and cancels all other tasks that have not completed yet, for example:

String bulb = consumerService.invokeAny(tasks);Copy

If you don't want to wait for all Future to finish, proceed as follows:

int queueSize = queue.size();  
List<Future<String>> futures = new ArrayList<>();  
for (int i = 0; i < queueSize; i++) {  
 futures.add(consumerService.submit(consumer));  
}  
  
for (Future<String> future: futures) {  
 String bulb = future.get();  
 logger.info(() -> "Future done: " + bulb);  
}Copy

This will not block until all tasks are done. Take a look at the following output sample:

[12:08:56] [INFO ] Packed: bulb-894 by consumer: ForkJoinPool-1-worker-7  
[12:08:56] [INFO ] Future done: bulb-894  
[12:08:56] [INFO ] Packed: bulb-953 by consumer: ForkJoinPool-1-worker-5  
...Copy

208. Latches

A *latch* is a Java synchronizer that allows one or more threads to wait until a bunch of events in other threads has completed. It starts from a given counter (commonly representing the number of events that should be waited), and each event that completes is responsible for decrementing the counter. When the counter reaches zero all the waiting threads can pass through. This is the terminal state of a latch. A latch cannot be reset or reused, so the waited events can happen only once. The following diagram shows, in four steps, how a latch with three threads works:

Imagen que contiene objeto, reloj, grupo

Descripción generada automáticamente

In API terms, a latch is implemented using java.util.concurrent.CountDownLatch.

The initial counter is set in the CountDownLatch constructor as an integer. For example, a CountDownLatch with a counter equal to 3 can be defined as follows:

CountDownLatch latch = new CountDownLatch(3);Copy

All threads that call the await() method will be blocked until the counter reaches zero. So, a thread that wants to be blocked until the latch reaches the terminal state will call await(). Each event that completes can call the countDown() method. This method decrements the counter with one value. Until the counter becomes zero, the threads that called await() are still blocked.

A latch can be used for a wide range of problems. For now, let's focus on our problem that should simulate the process of starting a server. The server is considered started after its internal services have started. Services can be started concurrently and are independent of each other. Starting a server is a process that takes a while and requires us to start all the underlying services of that server. Therefore, the thread that finalizes and validates the server start should wait until all server services (events) have started in other threads. If we assume that we have three services, we can write a ServerService class as follows:

public class ServerInstance implements Runnable {  
  
 private static final Logger logger =  
 Logger.getLogger(ServerInstance.class.getName());  
  
 private final CountDownLatch latch = new CountDownLatch(3);  
  
 @Override  
 public void run() {  
 logger.info("The server is getting ready to start ");  
 logger.info("Starting services ...\n");  
  
 long starting = System.currentTimeMillis();  
  
 Thread service1 = new Thread(  
 new ServerService(latch, "HTTP Listeners"));  
 Thread service2 = new Thread(  
 new ServerService(latch, "JMX"));  
 Thread service3 = new Thread(  
 new ServerService(latch, "Connectors"));  
  
 service1.start();  
 service2.start();  
 service3.start();  
  
 try {  
 latch.await();  
 logger.info(() -> "Server has successfully started in "   
 + (System.currentTimeMillis() - starting) / 1000   
 + " seconds");  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 // log ex  
 }  
 }  
}Copy

First, we define a CountDownLatch with a counter of three. Second, we start the services in three different threads. Finally, we block this thread via await(). Now, the following class simulates the starting process of services via random sleep:

public class ServerService implements Runnable {  
  
 private static final Logger logger =  
 Logger.getLogger(ServerService.class.getName());  
  
 private final String serviceName;  
 private final CountDownLatch latch;  
 private final Random rnd = new Random();  
  
 public ServerService(CountDownLatch latch, String serviceName) {  
 this.latch = latch;  
 this.serviceName = serviceName;  
 }  
  
 @Override  
 public void run() {  
  
 int startingIn = rnd.nextInt(10) \* 1000;  
  
 try {  
 logger.info(() -> "Starting service '" + serviceName + "' ...");  
  
 Thread.sleep(startingIn);  
  
 logger.info(() -> "Service '" + serviceName   
 + "' has successfully started in "   
 + startingIn / 1000 + " seconds");  
  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 // log ex  
 } finally {  
 latch.countDown();  
  
 logger.info(() -> "Service '" + serviceName + "' running ...");  
 }  
 }  
}Copy

Each service that started successfully (or failed) will decrement the latch via countDown(). Once the counter reaches zero, the server is considered started. Let's call it:

Thread server = new Thread(new ServerInstance());  
server.start();Copy

Here is a possible output:

**[08:49:17] [INFO] The server is getting ready to start**  
  
**[08:49:17] [INFO] Starting services ...**  
**[08:49:17] [INFO] Starting service 'JMX' ...**  
**[08:49:17] [INFO] Starting service 'Connectors' ...**  
**[08:49:17] [INFO] Starting service 'HTTP Listeners' ...**  
  
**[08:49:22] [INFO] Service 'HTTP Listeners' started in 5 seconds**  
**[08:49:22] [INFO] Service 'HTTP Listeners' running ...**  
**[08:49:25] [INFO] Service 'JMX' started in 8 seconds**  
**[08:49:25] [INFO] Service 'JMX' running ...**  
**[08:49:26] [INFO] Service 'Connectors' started in 9 seconds**  
**[08:49:26] [INFO] Service 'Connectors' running ...**  
  
**[08:49:26] [INFO] Server has successfully started in 9 seconds**Copy

In order to avoid indefinite waiting, the CountDownLatch class has an await() flavor that accepts a timeout, await​(long timeout, TimeUnit unit). If the waiting time elapses before the count reaches zero, this method returns false.

209. Barrier

A *barrier* is a Java synchronizer that allows a group of threads (known as *parties*) to reach a common barrier point. Basically, a group of threads waits for each other to meet at the barrier. It is like a bunch of friends who decide on a meeting point, and when all of them get this point, they go farther together. They won't leave the meeting point until all of them have arrived or until they feel they've been waiting too long.

This synchronizer works well for problems that rely on a task that can be divided into subtasks. Each subtask runs in a different thread and waits for the rest of the threads. When all the threads complete, they combine their results in a single result.

This following diagram shows an example of a barrier flow with three threads:

Diagrama, Esquemático

Descripción generada automáticamente

In API terms, a barrier is implemented using java.util.concurrent.CyclicBarrier.

A CyclicBarrier can be constructed via two constructors:

* One of them allows us to specify the number of parties (this is an integer)
* The other one allows us to add an action that should take place after all parties are at the barrier (this is a Runnable)

This action takes place when all threads in the party arrive, but before the release of any threads.

When a thread is ready to wait at the barrier, it simply calls the await() method. This method can wait indefinitely or until the specified timeout (if the specified timeout elapses or the thread is interrupted, this thread is released with a TimeoutException; the barrier is considered *broken*, and all the waiting threads at the barrier are released with a BrokenBarrierException). We can find out how many parties are required to trip this barrier via the getParties() method and how many are currently waiting at the barrier via the getNumberWaiting() method.

The await() method returns an integer that represents the arrival index of the current thread, where the index getParties()— 1 or 0 indicates the first or the last to arrive, respectively.

Let's assume that we want to start a server. The server is considered started after its internal services have started. Services can be prepared for start concurrently (this is time-consuming), but they run interdependently, therefore, once they are ready to start, they must be started all at once.

So, each service can be prepared to start in a separate thread. Once it is ready to start, the thread will wait at the barrier for the rest of the services. When all of them are ready to start, they cross the barrier and start running. Let's consider three services, so CyclicBarrier can be defined as follows:

Runnable barrierAction  
 = () -> logger.info("Services are ready to start ...");  
  
CyclicBarrier barrier = new CyclicBarrier(3, barrierAction);Copy

And, let's prepare the services via three threads:

public class ServerInstance implements Runnable {  
  
 private static final Logger logger  
 = Logger.getLogger(ServerInstance.class.getName());  
  
 private final Runnable barrierAction  
 = () -> logger.info("Services are ready to start ...");  
  
 private final CyclicBarrier barrier   
 = new CyclicBarrier(3, barrierAction);  
  
 @Override  
 public void run() {  
 logger.info("The server is getting ready to start ");  
 logger.info("Starting services ...\n");  
  
 long starting = System.currentTimeMillis();  
  
 Thread service1 = new Thread(  
 new ServerService(barrier, "HTTP Listeners"));  
 Thread service2 = new Thread(  
 new ServerService(barrier, "JMX"));  
 Thread service3 = new Thread(  
 new ServerService(barrier, "Connectors"));  
  
 service1.start();  
 service2.start();  
 service3.start();  
  
 try {  
 service1.join();  
 service2.join();  
 service3.join();  
  
 logger.info(() -> "Server has successfully started in "   
 + (System.currentTimeMillis() - starting) / 1000   
 + " seconds");  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 }  
 }  
}Copy

The ServerService is responsible for preparing each service to start and blocking it at the barrier via await():

public class ServerService implements Runnable {  
  
 private static final Logger logger =  
 Logger.getLogger(ServerService.class.getName());  
  
 private final String serviceName;  
 private final CyclicBarrier barrier;  
 private final Random rnd = new Random();  
  
 public ServerService(CyclicBarrier barrier, String serviceName) {  
 this.barrier = barrier;  
 this.serviceName = serviceName;  
 }  
  
 @Override  
 public void run() {  
  
 int startingIn = rnd.nextInt(10) \* 1000;  
  
 try {  
 logger.info(() -> "Preparing service '"   
 + serviceName + "' ...");  
  
 Thread.sleep(startingIn);  
 logger.info(() -> "Service '" + serviceName   
 + "' was prepared in " + startingIn / 1000   
 + " seconds (waiting for remaining services)");  
  
 barrier.await();  
  
 logger.info(() -> "The service '" + serviceName   
 + "' is running ...");  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 } catch (BrokenBarrierException ex) {  
 logger.severe(() -> "Exception ... barrier is broken! " + ex);  
 }  
 }  
}Copy

Now, let's run it:

Thread server = new Thread(new ServerInstance());  
server.start();Copy

Here is a possible output (notice how the threads have been released to cross the barrier):

[10:38:34] [INFO] The server is getting ready to start  
  
[10:38:34] [INFO] Starting services ...  
[10:38:34] [INFO] Preparing service 'Connectors' ...  
[10:38:34] [INFO] Preparing service 'JMX' ...  
[10:38:34] [INFO] Preparing service 'HTTP Listeners' ...  
  
[10:38:35] [INFO] Service 'HTTP Listeners' was prepared in 1 seconds  
 (waiting for remaining services)  
[10:38:36] [INFO] Service 'JMX' was prepared in 2 seconds  
 (waiting for remaining services)  
[10:38:38] [INFO] Service 'Connectors' was prepared in 4 seconds  
 (waiting for remaining services)  
  
[10:38:38] [INFO] Services are ready to start ...  
  
[10:38:38] [INFO] The service 'Connectors' is running ...  
[10:38:38] [INFO] The service 'HTTP Listeners' is running ...  
[10:38:38] [INFO] The service 'JMX' is running ...  
  
[10:38:38] [INFO] Server has successfully started in 4 secondsCopy

A CyclicBarrier is cyclic because it can be reset and reused. For this, call the reset() method after all threads waiting at the barrier are released, otherwise BrokenBarrierException will be thrown.  
  
A barrier that is in a *broken* state will cause the isBroken() flag method to return true.

210. Exchanger

An *exchanger* is a Java synchronizer that allows two threads to exchange objects at an exchange or synchronization point.

Mainly, this kind of synchronizer acts as a barrier. Two threads wait for each other at a barrier. They exchange an object and continue their usual tasks when both arrive.

The following diagram depicts in four steps the flow of an exchanger:

Diagrama

Descripción generada automáticamente

In API terms, this synchronizer is exposed by java.util.concurrent.Exchanger.

An Exchanger can be created via an empty constructor and exposes two exchange() methods:

* One that gets only the object that it will offer
* One that gets a timeout (before another thread enters the exchange, if the specified waiting time elapses, a TimeoutException will be thrown).

Remember our assembly line for bulbs? Well, let's assume that the producer (checker) adds the checked bulbs into a basket (for example, List<String>). When the basket is full, the producer exchanges it with the consumer (the packer) for an empty basket (for example, another List<String>). The process repeats as long as the assembly line is running.

The following diagram represents this flow:

Diagrama

Descripción generada automáticamente

So, first we need the Exchanger:

private static final int BASKET\_CAPACITY = 5;  
...  
private static final Exchanger<List<String>> exchanger   
 = new Exchanger<>();Copy

The producer fills up the basket and waits at the exchanging point for the consumer:

private static final int MAX\_PROD\_TIME\_MS = 2 \* 1000;  
private static final Random rnd = new Random();  
private static volatile boolean runningProducer;  
...  
private static class Producer implements Runnable {  
  
 private List<String> basket = new ArrayList<>(BASKET\_CAPACITY);  
  
 @Override  
 public void run() {  
  
 while (runningProducer) {  
 try {  
 for (int i = 0; i < BASKET\_CAPACITY; i++) {  
  
 String bulb = "bulb-" + rnd.nextInt(1000);  
 Thread.sleep(rnd.nextInt(MAX\_PROD\_TIME\_MS));  
 basket.add(bulb);  
  
 logger.info(() -> "Checked and added in the basket: "   
 + bulb);  
 }  
  
 logger.info("Producer: Waiting to exchange baskets ...");  
  
 basket = exchanger.exchange(basket);  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 break;  
 }  
 }  
 }  
}Copy

On the other hand, the consumer waits at the exchanging point to receive the basket full of bulbs from the producer and gives an empty one in exchange. Further, while the producer fills up the basket again, the consumer packs the bulbs from the received basket. When they are finished, they will go to the exchange point again to wait for another full basket. So, Consumer can be written as follows:

private static final int MAX\_CONS\_TIME\_MS = 5 \* 1000;  
private static final Random rnd = new Random();  
private static volatile boolean runningConsumer;  
...  
private static class Consumer implements Runnable {  
  
 private List<String> basket = new ArrayList<>(BASKET\_CAPACITY);  
  
 @Override  
 public void run() {  
  
 while (runningConsumer) {  
 try {  
 logger.info("Consumer: Waiting to exchange baskets ...");  
 basket = exchanger.exchange(basket);  
 logger.info(() -> "Consumer: Received the following bulbs: "   
 + basket);  
  
 for (String bulb: basket) {  
 if (bulb != null) {  
 Thread.sleep(rnd.nextInt(MAX\_CONS\_TIME\_MS));  
 logger.info(() -> "Packed from basket: " + bulb);  
 }  
 }  
  
 basket.clear();  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 break;  
 }  
 }  
 }  
}Copy

The rest of the code was omitted for brevity.

Now, let's see a possible output:

Starting assembly line ...  
[13:23:13] [INFO] Consumer: Waiting to exchange baskets ...  
[13:23:15] [INFO] Checked and added in the basket: bulb-606  
...  
[13:23:18] [INFO] Producer: Waiting to exchange baskets ...  
[13:23:18] [INFO] Consumer: Received the following bulbs:  
[bulb-606, bulb-251, bulb-102, bulb-454, bulb-280]  
[13:23:19] [INFO] Checked and added in the basket: bulb-16  
...  
[13:23:21] [INFO] Packed from basket: bulb-606  
...Copy

211. Semaphores

A *semaphore* is a Java synchronizer that allows us to control the number of threads that can access a resource at any one time. Conceptually, this synchronizer manages a bunch of *permits* (for example, similar to tokens). A thread that needs access to the resource must acquire a permit from the synchronizer. After the thread finishes its job with the resource, it must release the permit by returning it to the semaphore so that another thread can acquire it. A thread can acquire a permit immediately (if a permit is free), can wait for a certain amount of time, or can wait until a permit becomes free. Moreover, a thread can acquire and release more than one permit at a time, and a thread can release a permit even if it did not acquire one. This will add a permit to the semaphore; therefore a semaphore can start with one number of permits and die with another.

In API terms, this synchronizer is represented by java.util.concurrent.Semaphore.

Creating a Semaphore is as easy as calling one of its two constructors:

* public Semaphore​(int permits)
* public Semaphore​(int permits, boolean fair)

A fair Semaphore guarantees FIFO granting of permits under contention.

Acquiring a permit can be accomplished using the acquire() method. The process can be represented by the following bullets:

* Without arguments, this method will acquire a permit from this semaphore, blocking until one is available, or the thread is interrupted
* To acquire more than one permit, use acquire​(int permits)
* To try to acquire a permit and return a flag value immediately, use tryAcquire() or tryAcquire​(int permits)
* To acquire a permit by waiting for one to become available within the given waiting time (and the current thread has not been interrupted), use tryAcquire​(int permits, long timeout, TimeUnit unit)
* To acquire a permit from this semaphore, blocking until one is available can be obtained via acquireUninterruptibly() and acquireUninterruptibly(int permits)
* To release a permit, use release()

Now, in our scenario, a barbershop has three seats and serves the customers in a FIFO manner. A customer tries for five seconds to take a seat. In the end, it releases the acquired seat. Check out the following code to see how a seat can be acquired and released:

public class Barbershop {  
  
 private static final Logger logger =  
 Logger.getLogger(Barbershop.class.getName());  
  
 private final Semaphore seats;  
  
 public Barbershop(int seatsCount) {  
 this.seats = new Semaphore(seatsCount, true);  
 }  
  
 public boolean acquireSeat(int customerId) {  
 logger.info(() -> "Customer #" + customerId   
 + " is trying to get a seat");  
  
 try {  
 boolean acquired = seats.tryAcquire(  
 5 \* 1000, TimeUnit.MILLISECONDS);  
  
 if (!acquired) {  
 logger.info(() -> "Customer #" + customerId   
 + " has left the barbershop");  
  
 return false;  
 }  
  
 logger.info(() -> "Customer #" + customerId + " got a seat");  
  
 return true;  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 }  
  
 return false;  
 }  
  
 public void releaseSeat(int customerId) {  
 logger.info(() -> "Customer #" + customerId   
 + " has released a seat");  
 seats.release();  
 }  
}Copy

If no seat has been freed in these five seconds, the person leaves the barber shop. On the other hand, a customer that succeeds in taking a seat is served by a barber (this will take a random number of seconds between 0 and 10). Finally, the customer releases the seat. In code lines, this can be written as follows:

public class BarbershopCustomer implements Runnable {  
  
 private static final Logger logger =  
 Logger.getLogger(BarbershopCustomer.class.getName());  
 private static final Random rnd = new Random();  
  
 private final Barbershop barbershop;  
 private final int customerId;  
  
 public BarbershopCustomer(Barbershop barbershop, int customerId) {  
 this.barbershop = barbershop;  
 this.customerId = customerId;  
 }  
  
 @Override  
 public void run() {  
  
 boolean acquired = barbershop.acquireSeat(customerId);  
  
 if (acquired) {  
 try {  
 Thread.sleep(rnd.nextInt(10 \* 1000));  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 } finally {  
 barbershop.releaseSeat(customerId);  
 }  
 } else {  
 Thread.currentThread().interrupt();  
 }  
 }  
}Copy

Let's bring 10 customers to our barbershop:

Barbershop bs = new Barbershop(3);  
  
for (int i = 1; i <= 10; i++) {  
 BarbershopCustomer bc = new BarbershopCustomer(bs, i);  
 new Thread(bc).start();  
}Copy

Here is a snapshot of a possible output:

[16:36:17] [INFO] Customer #10 is trying to get a seat  
[16:36:17] [INFO] Customer #5 is trying to get a seat  
[16:36:17] [INFO] Customer #7 is trying to get a seat  
[16:36:17] [INFO] Customer #5 got a seat  
[16:36:17] [INFO] Customer #10 got a seat  
[16:36:19] [INFO] Customer #10 has released a seat  
...Copy

**A permit is not acquired on a thread basis.**  
  
This means that the T1 thread can acquire a permit from a Semaphore and the T2 thread can release it. Of course, the developer is responsible for managing the process.

212. Phasers

A *phaser* is a flexible Java synchronizer that combines the functionalities of CyclicBarrier and CountDownLatch in the following context:

* A phaser is made of one or multiple phases that act as barriers for a dynamic number of parties (threads).
* During a phaser lifespan, the number of synchronized parties (threads) can be modified dynamically. We can register/deregister parties.
* The currently-registered parties must wait in the current phase (barrier) before going to the next step of execution (next phase)—as in the case of CyclicBarrier.
* Each phase of a phaser can be identified via an associated number/index starting from 0. The first phase is 0, the next phase is 1, the next phase is 2, and so on until Integer.MAX\_VALUE.
* A phaser can have three types of parties in any of its phases: *registered*, *arrived* (these are registered parties waiting at the current phase/barrier), and *unarrived* (these are registered parties on the way to the current phase).
* There are three types of dynamic counters for parties: a counter for registered parties, a counter for arrived parties, and a counter for unarrived parties. When all parties arrive at the current phase (the number of registered parties is equal to the number of arrived parties), the phaser will advance to the next phase.
* Optionally, we can execute an action (snippet of code) right before advancing to the next phase (when all the parties arrive at the phase/barrier).
* A phaser has a termination state. Counts of registered parties are unaffected by termination, but after termination, all synchronization methods immediately return without waiting to advance to another phase. Similarly, attempts to register upon termination have no effect.

In the following diagram, we can see a phaser with four registered parties in phase 0, and three registered parties in phase 1. We also have some API flavors that are discussed further:

Diagrama, Esquemático

Descripción generada automáticamente

Commonly, by parties, we understand threads (one party = one thread), but a phaser doesn't perform an association between a party and a specific thread. A phaser just counts and manages the number of registered and deregistered parties.

In API terms, this synchronizer is represented by java.util.concurrent.Phaser.

A Phaser can be created with zero parties, an explicit number of parties via an empty constructor, or a constructor that takes an integer argument, Phaser​(int parties). A Phaser can also have a parent specified via Phaser​(Phaser parent) or Phaser​(Phaser parent, int parties). It's common to start a Phaser with a single party, known as the controller or control-party. Usually, this party lives the longest during the Phaser lifespan.

A party can be registered any time via the register() method (in the preceding diagram, between phase 0 and phase 1, we register **T5** and **T6**). We can also register a bulk of parties via bulkRegister​(int parties). A registered party can be deregistered without waiting for other parties via arriveAndDeregister(). This method allows a party to arrive at the current barrier (Phaser) and deregisters it without waiting for other parties to arrive (in the preceding diagram, the **T4**, **T3**, and **T2** parties are deregistered one by one). Each deregistered party decreases the number of registered parties by one.

In order to arrive at the current phase (barrier) and wait for other parties to arrive, we need to call the arriveAndAwaitAdvance() method. This method blocks until all registered parties arrive at the current phase. All parties will advance to the next phase of this Phaser once the last registered party arrives at the current phase.

Optionally, when all registered parties arrive at the current phase, we can run a specific action by overriding the onAdvance() method, onAdvance​(int phase, int registeredParties). This method returns a boolean value which is true if we want to trigger the termination of Phaser. In addition, we can force the termination via forceTermination(), and we can test it via the flag method, isTerminated(). Overriding the onAdvance() method requires us to extend the Phaser class (usually, via an anonymous class).

At this moment, we should have enough details to solve our problem. So, we have to simulate the start process of a server in three phases of a Phaser. The server is considered started and running after its five internal services have started. In the first phase, we need to concurrently start three services. In the second phase, we need to concurrently start two more services (these can be started only if the first three are already running). In phase three, the server performs a final check-in and is considered started and running.

So, the thread (party) that manages the server-starting process can be considered the thread that controls the rest of the threads (parties). This means that we can create the Phaser and register this control thread (or, controller) via the Phaser constructor:

public class ServerInstance implements Runnable {  
  
 private static final Logger logger =  
 Logger.getLogger(ServerInstance.class.getName());  
  
 private final Phaser phaser = new Phaser(1) {  
  
 @Override  
 protected boolean onAdvance(int phase, int registeredParties) {  
 logger.warning(() -> "Phase:" + phase   
 + " Registered parties: " + registeredParties);  
  
 return registeredParties == 0;  
 }  
 };  
 ...  
}Copy

Using an anonymous class, we create this Phaser object and override its onAdvance() method to define an action that has two main purposes:

* Print a quick status of the current phase and number of registered parties
* If there are no more registered parties, trigger the Phaser termination

This method will be called for every phase when all the currently-registered parties arrive at the current barrier (current phase).

The threads that manage the server's services need to start these services and to deregister themselves from the Phaser. So, each service is started in a separate thread that will deregister at the end of its job via arriveAndDeregister(). For this, we can use the following Runnable:

public class ServerService implements Runnable {  
  
 private static final Logger logger =  
 Logger.getLogger(ServerService.class.getName());  
  
 private final String serviceName;  
 private final Phaser phaser;  
 private final Random rnd = new Random();  
  
 public ServerService(Phaser phaser, String serviceName) {  
 this.phaser = phaser;  
 this.serviceName = serviceName;  
 this.phaser.register();  
 }  
  
 @Override  
 public void run() {  
  
 int startingIn = rnd.nextInt(10) \* 1000;  
  
 try {  
 logger.info(() -> "Starting service '" + serviceName + "' ...");  
 Thread.sleep(startingIn);  
 logger.info(() -> "Service '" + serviceName   
 + "' was started in " + startingIn / 1000   
 + " seconds (waiting for remaining services)");  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 } finally {  
 phaser.arriveAndDeregister();  
 }  
 }  
}Copy

Now, the control thread can trigger the start process for service1, service2, and service3. This process is shaped in the following method:

private void startFirstThreeServices() {  
  
 Thread service1 = new Thread(  
 new ServerService(phaser, "HTTP Listeners"));  
 Thread service2 = new Thread(  
 new ServerService(phaser, "JMX"));  
 Thread service3 = new Thread(  
 new ServerService(phaser, "Connectors"));  
  
 service1.start();  
 service2.start();  
 service3.start();  
  
 phaser.arriveAndAwaitAdvance(); // phase 0  
}Copy

Notice that, at the end of this method, we call phaser.arriveAndAwaitAdvance(). This is the control-party that waits for the rest of the registered parties to arrive. The rest of the registered parties (service1, service2, and service3) are deregistered one by one until the control-party is the only one left in Phaser. At this point, it's time to advance to the next phase. So, the control-party is the only one that advances to the next phase.

Similar to this implementation, the control thread can trigger the start process for service4 and service5. This process is shaped in the following method:

private void startNextTwoServices() {  
  
 Thread service4 = new Thread(  
 new ServerService(phaser, "Virtual Hosts"));  
 Thread service5 = new Thread(  
 new ServerService(phaser, "Ports"));  
  
 service4.start();  
 service5.start();  
  
 phaser.arriveAndAwaitAdvance(); // phase 1  
}Copy

Finally, after these five services are started, the control thread performs one last check that was implemented in the following method as a dummy Thread.sleep(). Notice that, at the end of this action, the control thread that has started the server deregistered itself from the Phaser. When this happens, it means there are no more registered parties and the Phaser is terminated as a result of returning true from the onAdvance() method:

private void finalCheckIn() {  
  
 try {  
 logger.info("Finalizing process (should take 2 seconds) ...");  
 Thread.sleep(2000);  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 } finally {  
 phaser.arriveAndDeregister(); // phase 2  
 }  
}Copy

The job of the control thread is to call the preceding three methods in the proper order. The rest of the code consists of some logs; therefore it was skipped for brevity. The complete source code of this problem is bundled with this book.

At any time, we can find out the number of registered parties via getRegisteredParties(), the number of arrived parties via getArrivedParties(), and the number of unarrived parties via getUnarrivedParties(). You might also want to check the arrive(), awaitAdvance​(int phase), and awaitAdvanceInterruptibly​(int phase) methods.

Summary

This chapter outlined the main coordinates of Java concurrency and should prepare you for the next chapter. We covered several fundamental problems about thread life cycles, object- and class-level locking, thread pools, and Callable and Future.

Download the applications from this chapter to see the results and to check out some additional details.

Concurrency - Deep Dive

This chapter includes 13 problems that involve Java concurrency, covering areas such as the fork/join framework, CompletableFuture, ReentrantLock, ReentrantReadWriteLock, StampedLock, atomic variables, tasks cancellation, interruptible methods, thread-local, and deadlocks. Concurrency is one of the required topics for any developer and can't be ignored at a job interview. That's why this chapter and the last one are so important. On finishing this chapter, you'll have a considerable understanding of concurrency, which every Java developer needs.

Problems

Use the following problems to test your concurrency programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Interruptible methods**:Write a program that exemplifies the best approach for dealing with an interruptible method.
2. **Fork/join framework**: Write a program that relies on the fork/join framework to sum the elements of a list. Write a program that relies on the fork/join framework to compute the Fibonacci number at a given position (for example, F12 = 144). In addition, write a program that exemplifies the usage of CountedCompleter.
3. **Fork/join framework and compareAndSetForkJoinTaskTag()**: Write a program that applies the fork/join framework to a suite of interdependent tasks that should be executed only once (for example, *task D* depends on *task C* and *task B*, but *task C* depends on *task B* as well; therefore, *task B* must be executed only once, not twice).
4. **CompletableFuture**: Write several snippets of code to exemplify asynchronous code via CompletableFuture.
5. **Combining multiple CompletableFuture** **objects**: Write several snippets of code to exemplify different solutions for combining multiple CompletableFuture objects together.
6. **Optimizing busy waiting**: Write a proof of concept to exemplify the optimization of a *busy waiting* technique via onSpinWait().
7. **Task cancellation**: Write a proof of concept that exemplifies the usage of a volatile variable for holding the cancellation state of a process.
8. ThreadLocal: Write a proof of concept that exemplifies the usage of ThreadLocal.
9. **Atomic variables**: Write a program that counts the integers from 1 to 1,000,000 using a multithreaded application (Runnable).
10. **ReentrantLock**: Write a program that increments the integers from 1 to 1,000,000 using ReentrantLock.
11. **ReentrantReadWriteLock**: Write a program that simulates the orchestration of a read-write process via ReentrantReadWriteLock.
12. **StampedLock**: Write a program that simulates the orchestration of a read-write process via StampedLock.
13. **Deadlock (dining philosophers)**: Write a program that reveals and solves the deadlock (*circular wait* or *deadly embrace*) that may occur in the famous dining philosophers problem.

Solutions

The following sections describe solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations shown here include only the most interesting and important details needed to solve the problems. Download the example solutions to see additional details and to experiment with the programs at <https://github.com/PacktPublishing/Java-Coding-Problems>.

213. Interruptible methods

By an interruptible method, we mean a blocking method that may throw InterruptedException, for example, Thread.sleep(), BlockingQueue.take(), BlockingQueue.poll(long timeout, TimeUnit unit), and so on. A blocking thread is usually in a **BLOCKED**, **WAITING**, or **TIMED\_WAITING** state, and, if it is interrupted, then the method tries to throw InterruptedException as soon as possible.

Since InterruptedException is a checked exception, we must catch it and/or throw it. In other words, if our method calls a method that throws InterruptedException, then we must be prepared to deal with this exception. If we can throw it (propagate the exception to the caller), then it is not our job anymore. The caller has to deal with it further. So, let's focus on the case when we must catch it. Such a case can occur when our code is run inside Runnable, which cannot throw an exception.

Let's start with a simple example. Trying to get an element from BlockingQueue via poll(long timeout, TimeUnit unit) can be written as follows:

try {  
 queue.poll(3000, TimeUnit.MILLISECONDS);  
} catch (InterruptedException ex) {  
 ...  
 logger.info(() -> "Thread is interrupted? "  
 + Thread.currentThread().isInterrupted());  
}Copy

Attempting to poll an element from the queue can result in InterruptedException. There is a window of 3,000 milliseconds in which the thread can be interrupted. In case of an interruption (for example, Thread.interrupt()), we may be tempted to think that calling Thread.currentThread().isInterrupted() in the catch block will return true. After all, we are in an InterruptedException catch block, so it makes sense to believe this. Actually, it will return false, and the answer is in the source code of the poll(long timeout, TimeUnit unit) method listed as follows:

1: public E poll(long timeout, TimeUnit unit)   
 throws InterruptedException {  
2: E e = xfer(null, false, TIMED, unit.toNanos(timeout));  
3: if (e != null || !Thread.interrupted())  
4: return e;  
5: throw new InterruptedException();  
6: }Copy

More precisely, the answer is in line 3. If the thread was interrupted then Thread.interrupted() will return true and will lead to line 5 (throw new InterruptedException()). But beside testing, if the current thread was interrupted, Thread.interrupted() clears the interrupted status of the thread. Check out the following succession of calls for an interrupted thread:

Thread.currentThread().isInterrupted(); // true  
Thread.interrupted() // true  
Thread.currentThread().isInterrupted(); // false  
Thread.interrupted() // falseCopy

Notice that Thread.currentThread().isInterrupted() tests whether this thread has been interrupted without affecting the interrupted status.

Now, let's get back to our case. So, we know that the thread was interrupted since we caught InterruptedException, but the interrupted status was cleared by Thread.interrupted(). This means also that the caller of our code will not be aware of the interruption.

It is our responsibility to be good citizens and restore the interrupt by calling the interrupt() method. This way, the caller of our code can see that an interrupt was issued and act accordingly. The correct code could be as follows:

try {  
 queue.poll(3000, TimeUnit.MILLISECONDS);  
} catch (InterruptedException ex) {  
 ...  
 Thread.currentThread().interrupt(); // restore interrupt  
}Copy

As a rule of thumb, after catching InterruptedException, do not forget to restore the interrupt by calling Thread.currentThread().interrupt().

Let's tackle a problem that highlights the case of forgetting to restore the interrupt. Let's assume a Runnable that runs as long as the current thread is not interrupted (for example, while (!Thread.currentThread().isInterrupted()) { ... }).

At each iteration, if the current thread interrupted status is false, then we try to get an element from BlockingQueue.

The following code is the implementation:

Thread thread = new Thread(() -> {  
  
 // some dummy queue  
 TransferQueue<String> queue = new LinkedTransferQueue<>();  
  
 while (!Thread.currentThread().isInterrupted()) {  
 try {  
 logger.info(() -> "For 3 seconds the thread "   
 + Thread.currentThread().getName()   
 + " will try to poll an element from queue ...");  
  
 queue.poll(3000, TimeUnit.MILLISECONDS);  
 } catch (InterruptedException ex) {  
 logger.severe(() -> "InterruptedException! The thread "  
 + Thread.currentThread().getName() + " was interrupted!");  
 Thread.currentThread().interrupt();  
 }  
 }  
  
 logger.info(() -> "The execution was stopped!");  
});Copy

As a caller (another thread), we start the above thread, sleep for 1.5 seconds, just to give time to this thread to enter in the poll() method, and we interrupt it. This is shown in the following code:

thread.start();  
Thread.sleep(1500);  
thread.interrupt();Copy

This will lead to InterruptedException.

The exception is logged and the interrupt is restored.

At the next step, while evaluates Thread.currentThread().isInterrupted() to false and exits.

As a result, the output will be as follows:

[18:02:43] [INFO] For 3 seconds the thread Thread-0  
 will try to poll an element from queue ...  
  
[18:02:44] [SEVERE] InterruptedException!  
 The thread Thread-0 was interrupted!  
  
[18:02:45] [INFO] The execution was stopped!Copy

Now, let's comment on the line that restores the interrupt:

...  
} catch (InterruptedException ex) {  
 logger.severe(() -> "InterruptedException! The thread "   
 + Thread.currentThread().getName() + " was interrupted!");  
  
 // notice that the below line is commented  
 // Thread.currentThread().interrupt();  
}  
...Copy

This time, the while block will run forever since its guarding condition is always evaluated to true.

The code cannot act on the interruption, so the output will be as follows:

[18:05:47] [INFO] For 3 seconds the thread Thread-0  
 will try to poll an element from queue ...  
  
[18:05:48] [SEVERE] InterruptedException!  
 The thread Thread-0 was interrupted!  
  
[18:05:48] [INFO] For 3 seconds the thread Thread-0  
 will try to poll an element from queue ...  
...Copy

As a rule of thumb, the only acceptable case when we can swallow an interrupt (not restore the interrupt) is when we can control the entire call stack (for example, extend Thread).  
  
Otherwise, catching InterruptedException should contain Thread.currentThread().interrupt() as well.

214. Fork/join framework

We've already had an introduction to the fork/join framework in the *Work-stealing thread pool* section.

Mainly, the fork/join framework is meant to take a big task (typically, by big we understand a large volume of data) and recursively split (fork) it into smaller tasks (subtasks) that can be performed in parallel. In the end, after all the subtasks have been completed, their results are combined (joined) in a single result.

The following diagram is a visual representation of a fork-join flow:

Diagrama

Descripción generada automáticamente

In API terms, a fork/join can be created via java.util.concurrent.ForkJoinPool.

Before JDK 8, the recommended approach relied on a public static variable as follows:

public static ForkJoinPool forkJoinPool = new ForkJoinPool();Copy

Starting with JDK 8, we can do it as follows:

ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();Copy

Both approaches avoid the unpleasant situation of having too many pool threads on a single JVM, caused by the parallel operations that created their own pools.

For a custom ForkJoinPool, rely on the constructors of this class. JDK 9 has added the most comprehensive one so far (details are available in the documentation).

A ForkJoinPool object manipulates tasks. The base type of task executed in ForkJoinPool is ForkJoinTask<V>. More precisely, the following tasks are executed:

* RecursiveAction for the void tasks
* RecursiveTask<V> for tasks that return a value
* CountedCompleter<T> for tasks that need to remember the pending task count

All three types of tasks have an abstract method named compute() in which the task's logic is shaped.

Submitting tasks to ForkJoinPool can be accomplished via the following:

* execute() and submit()
* invoke() for forking the task and waiting for the result
* invokeAll() for forking a bunch of tasks (for example, a collection)
* fork() for arranging to asynchronously execute this task in the pool, and join() for returning the result of the computation when it is done

Let's start with a problem solved via RecursiveTask.

Computing the sum via RecursiveTask

To demonstrate the forking behavior of the framework, let's assume that we have a list of numbers and we want to compute the sum of these numbers. For this, we recursively split (fork) this list as long as it is larger than the specified THRESHOLD using the createSubtasks() method. Each task is added into List<SumRecursiveTask>. In the end, this list is submitted to ForkJoinPool via the invokeAll​(Collection<T> tasks) method. This is done using the following code:

public class SumRecursiveTask extends RecursiveTask<Integer> {  
  
 private static final Logger logger   
 = Logger.getLogger(SumRecursiveTask.class.getName());  
 private static final int THRESHOLD = 10;  
  
 private final List<Integer> worklist;  
  
 public SumRecursiveTask(List<Integer> worklist) {  
 this.worklist = worklist;  
 }  
  
 @Override  
 protected Integer compute() {  
 if (worklist.size() <= THRESHOLD) {  
 return partialSum(worklist);  
 }  
  
 return ForkJoinTask.invokeAll(createSubtasks())  
 .stream()  
 .mapToInt(ForkJoinTask::join)  
 .sum();  
 }  
  
 private List<SumRecursiveTask> createSubtasks() {  
  
 List<SumRecursiveTask> subtasks = new ArrayList<>();  
 int size = worklist.size();  
  
 List<Integer> worklistLeft   
 = worklist.subList(0, (size + 1) / 2);  
 List<Integer> worklistRight   
 = worklist.subList((size + 1) / 2, size);  
  
 subtasks.add(new SumRecursiveTask(worklistLeft));  
 subtasks.add(new SumRecursiveTask(worklistRight));  
  
 return subtasks;  
 }  
  
 private Integer partialSum(List<Integer> worklist) {  
  
 int sum = worklist.stream()  
 .mapToInt(e -> e)  
 .sum();  
  
 logger.info(() -> "Partial sum: " + worklist + " = "  
 + sum + "\tThread: " + Thread.currentThread().getName());  
  
 return sum;  
 }  
}Copy

In order to test it, we need a list and ForkJoinPool as follows:

ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();  
  
Random rnd = new Random();  
List<Integer> list = new ArrayList<>();  
  
for (int i = 0; i < 200; i++) {  
 list.add(1 + rnd.nextInt(10));  
}  
  
SumRecursiveTask sumRecursiveTask = new SumRecursiveTask(list);  
Integer sumAll = forkJoinPool.invoke(sumRecursiveTask);  
  
logger.info(() -> "Final sum: " + sumAll);Copy

A possible output will be the following:

...  
[15:17:06] Partial sum: [1, 3, 6, 6, 2, 5, 9] = 32  
ForkJoinPool.commonPool-worker-9  
...  
[15:17:06] Partial sum: [1, 9, 9, 8, 9, 5] = 41  
ForkJoinPool.commonPool-worker-7  
[15:17:06] Final sum: 1084Copy

Computing Fibonacci via RecursiveAction

Commonly denoted as *Fn*, the Fibonacci numbers are a sequence that respects the following formula:

*F0=0, F1 = 1, ... Fn = Fn-1 + Fn-2 (n > 1)*

A snapshot of Fibonacci numbers is:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

The implementation of Fibonacci numbers via RecursiveAction can be accomplished as follows:

public class FibonacciRecursiveAction extends RecursiveAction {  
  
 private static final Logger logger =  
 Logger.getLogger(FibonacciRecursiveAction.class.getName());  
 private static final long THRESHOLD = 5;  
  
 private long nr;  
  
 public FibonacciRecursiveAction(long nr) {  
 this.nr = nr;  
 }  
  
 @Override  
 protected void compute() {  
  
 final long n = nr;  
  
 if (n <= THRESHOLD) {  
 nr = fibonacci(n);  
 } else {  
 nr = ForkJoinTask.invokeAll(createSubtasks(n))  
 .stream()  
 .mapToLong(x -> x.fibonacciNumber())  
 .sum();  
 }  
 }  
  
 private List<FibonacciRecursiveAction> createSubtasks(long n) {  
  
 List<FibonacciRecursiveAction> subtasks = new ArrayList<>();  
  
 FibonacciRecursiveAction fibonacciMinusOne  
 = new FibonacciRecursiveAction(n - 1);  
 FibonacciRecursiveAction fibonacciMinusTwo  
 = new FibonacciRecursiveAction(n - 2);  
  
 subtasks.add(fibonacciMinusOne);  
 subtasks.add(fibonacciMinusTwo);  
  
 return subtasks;  
 }  
  
 private long fibonacci(long n) {  
 logger.info(() -> "Number: " + n   
 + " Thread: " + Thread.currentThread().getName());  
  
 if (n <= 1) {  
 return n;  
 }  
  
 return fibonacci(n - 1) + fibonacci(n - 2);  
 }  
  
 public long fibonacciNumber() {  
 return nr;  
 }  
}Copy

In order to test it, we need the following ForkJoinPool object:

ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();  
  
FibonacciRecursiveAction fibonacciRecursiveAction  
 = new FibonacciRecursiveAction(12);  
forkJoinPool.invoke(fibonacciRecursiveAction);  
  
logger.info(() -> "Fibonacci: "  
 + fibonacciRecursiveAction.fibonacciNumber());Copy

The output for *F12*is as follows:

[15:40:46] Number: 5 Thread: ForkJoinPool.commonPool-worker-3  
[15:40:46] Number: 5 Thread: ForkJoinPool.commonPool-worker-13  
[15:40:46] Number: 4 Thread: ForkJoinPool.commonPool-worker-3  
[15:40:46] Number: 4 Thread: ForkJoinPool.commonPool-worker-9  
...  
[15:40:49] Number: 0 Thread: ForkJoinPool.commonPool-worker-7  
[15:40:49] Fibonacci: 144Copy

Using CountedCompleter

CountedCompleter is a type of ForkJoinTask added in JDK 8.

The job of CountedCompleter is to remember the pending task count (nothing less, nothing more). We can set the pending count via setPendingCount() or increment it with an explicit delta via addToPendingCount​(int delta). Commonly, we call these methods right before forking (for example, if we fork twice, then we call addToPendingCount(2) or setPendingCount(2), depending on the case).

In the compute() method, we decrease the pending count via tryComplete() or propagateCompletion(). When the tryComplete() method is called, with a pending count of zero, or the unconditional complete() method is called, the onCompletion() method is called. The propagateCompletion() method is similar with tryComplete(), but it doesn't call onCompletion().

CountedCompleter can optionally return a computed value. For this, we have to override the getRawResult() method to return a value.

The following code sums up all the values of a list via CountedCompleter:

public class SumCountedCompleter extends CountedCompleter<Long> {  
  
 private static final Logger logger   
 = Logger.getLogger(SumCountedCompleter.class.getName());  
 private static final int THRESHOLD = 10;  
 private static final LongAdder sumAll = new LongAdder();  
  
 private final List<Integer> worklist;  
  
 public SumCountedCompleter(  
 CountedCompleter<Long> c, List<Integer> worklist) {  
 super(c);  
 this.worklist = worklist;  
 }  
  
 @Override  
 public void compute() {  
 if (worklist.size() <= THRESHOLD) {  
 partialSum(worklist);  
 } else {  
 int size = worklist.size();  
  
 List<Integer> worklistLeft   
 = worklist.subList(0, (size + 1) / 2);  
 List<Integer> worklistRight   
 = worklist.subList((size + 1) / 2, size);  
  
 addToPendingCount(2);  
 SumCountedCompleter leftTask  
 = new SumCountedCompleter(this, worklistLeft);  
 SumCountedCompleter rightTask  
 = new SumCountedCompleter(this, worklistRight);  
  
 leftTask.fork();  
 rightTask.fork();  
 }  
  
 tryComplete();  
 }  
  
 @Override  
 public void onCompletion(CountedCompleter<?> caller) {  
 logger.info(() -> "Thread complete: "   
 + Thread.currentThread().getName());  
 }  
  
 @Override  
 public Long getRawResult() {  
 return sumAll.sum();  
 }  
  
 private Integer partialSum(List<Integer> worklist) {  
 int sum = worklist.stream()  
 .mapToInt(e -> e)  
 .sum();  
  
 sumAll.add(sum);  
  
 logger.info(() -> "Partial sum: " + worklist + " = "  
 + sum + "\tThread: " + Thread.currentThread().getName());  
  
 return sum;  
 }  
}Copy

Now, let's see a potential call and output:

ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();  
Random rnd = new Random();  
List<Integer> list = new ArrayList<>();  
  
for (int i = 0; i < 200; i++) {  
 list.add(1 + rnd.nextInt(10));  
}  
  
SumCountedCompleter sumCountedCompleter  
 = new SumCountedCompleter(null, list);  
forkJoinPool.invoke(sumCountedCompleter);  
  
logger.info(() -> "Done! Result: "  
 + sumCountedCompleter.getRawResult());Copy

The output will be as follows:

[11:11:07] Partial sum: [7, 7, 8, 5, 6, 10] = 43  
 ForkJoinPool.commonPool-worker-7  
[11:11:07] Partial sum: [9, 1, 1, 6, 1, 2] = 20  
 ForkJoinPool.commonPool-worker-3  
...  
[11:11:07] Thread complete: ForkJoinPool.commonPool-worker-15  
[11:11:07] Done! Result: 1159Copy

215. Fork/join framework and compareAndSetForkJoinTaskTag()

Now, that we are familiar with the fork/join framework, let's see another problem. This time let's assume that we have a suite of ForkJoinTask objects that are interdependent. The following diagram can be considered a use case:

Diagrama

Descripción generada automáticamente

Here is the description of the preceding diagram:

* **TaskD** has three dependencies: **TaskA**, **TaskB**, and **TaskC**.
* **TaskC** has two dependencies: **TaskA** and **TaskB**.
* **TaskB** has one dependency: **TaskA**.
* **TaskA** has no dependencies.

In code lines, we will shape it as follows:

ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();  
  
Task taskA = new Task("Task-A", new Adder(1));  
  
Task taskB = new Task("Task-B", new Adder(2), taskA);  
  
Task taskC = new Task("Task-C", new Adder(3), taskA, taskB);  
  
Task taskD = new Task("Task-D", new Adder(4), taskA, taskB, taskC);  
  
forkJoinPool.invoke(taskD);Copy

An Adder is a simple Callable that should be executed only once for each task (so, once for **TaskD**, **TaskC**, **TaskB**, and **TaskA**). The Adder is initiated in the following code:

private static class Adder implements Callable {  
  
 private static final AtomicInteger result = new AtomicInteger();  
  
 private Integer nr;  
  
 public Adder(Integer nr) {  
 this.nr = nr;  
 }  
  
 @Override  
 public Integer call() {  
 logger.info(() -> "Adding number: " + nr  
 + " by thread:" + Thread.currentThread().getName());  
  
 return result.addAndGet(nr);  
 }  
}Copy

We already know how to use the fork/join framework for tasks with acyclic and/or non-repeatable (or we don't care that they repeat) completion dependencies. But if we implement it this way then Callable will be called more than once per task. For example, **TaskA** appears as a dependency for three other tasks, so Callable will be invoked three times. We want it only once.

A very handy feature of ForkJoinPool added in JDK 8 consists of atomically tagging with a short value:

* short getForkJoinTaskTag(): Returns the tag for this task.
* short setForkJoinTaskTag​(short newValue): Atomically sets the tag value for this task and returns the old value.
* boolean compareAndSetForkJoinTaskTag​(short expect, short update): Returns true if the current value was equal to expect and was changed to update.

In other words, compareAndSetForkJoinTaskTag() allows us to tag a task as VISITED. Once it is tagged as VISITED, it will not be executed. Let's see it in the following code lines:

public class Task<Integer> extends RecursiveTask<Integer> {  
  
 private static final Logger logger   
 = Logger.getLogger(Task.class.getName());  
 private static final short UNVISITED = 0;  
 private static final short VISITED = 1;  
  
 private Set<Task<Integer>> dependencies = new HashSet<>();  
  
 private final String name;  
 private final Callable<Integer> callable;  
  
 public Task(String name, Callable<Integer> callable,  
 Task<Integer> ...dependencies) {  
 this.name = name;  
 this.callable = callable;  
 this.dependencies = Set.of(dependencies);  
 }  
  
 @Override  
 protected Integer compute() {  
 dependencies.stream()  
 .filter((task) -> (task.updateTaskAsVisited()))  
 .forEachOrdered((task) -> {  
 logger.info(() -> "Tagged: " + task + "("  
 + task.getForkJoinTaskTag() + ")");  
  
 task.fork();  
 });  
  
 for (Task task: dependencies) {  
 task.join();  
 }  
  
 try {  
 return callable.call();  
 } catch (Exception ex) {  
 logger.severe(() -> "Exception: " + ex);  
 }  
  
 return null;  
 }  
  
 public boolean updateTaskAsVisited() {  
 return compareAndSetForkJoinTaskTag(UNVISITED, VISITED);  
 }  
  
 @Override  
 public String toString() {  
 return name + " | dependencies=" + dependencies + "}";  
 }  
}Copy

And, a possible output could be the following:

[10:30:53] [INFO] Tagged: Task-B(1)  
[10:30:53] [INFO] Tagged: Task-C(1)  
[10:30:53] [INFO] Tagged: Task-A(1)  
[10:30:53] [INFO] Adding number: 1   
 by thread:ForkJoinPool.commonPool-worker-3  
[10:30:53] [INFO] Adding number: 2   
 by thread:ForkJoinPool.commonPool-worker-3  
[10:30:53] [INFO] Adding number: 3   
 by thread:ForkJoinPool.commonPool-worker-5  
[10:30:53] [INFO] Adding number: 4   
 by thread:main  
[10:30:53] [INFO] Result: 10Copy

216. CompletableFuture

JDK 8 has made a significant step forward in the world of asynchronous programming by enhancing Future with CompletableFuture. The main limitations of Future are:

* It cannot be explicitly complete.
* It doesn't support callbacks for performing actions on the result.
* They cannot be chained or combined for obtaining complex asynchronous pipelines.
* It doesn't provide exception handling.

A CompletableFuture doesn't have these limitations. A simple, but useless CompletableFuture can be written as follows:

CompletableFuture<Integer> completableFuture   
 = new CompletableFuture<>();Copy

The result can be obtained via the blocking get() method:

completableFuture.get();Copy

In addition to this, let's see several examples of running asynchronous tasks in the context of an e-commerce platform. We add these examples in a helper class named CustomerAsyncs.

Running asynchronous task and return void

User problem: *Print a certain customer order.*

Since printing is a process that doesn't need to return a result, this is a job for runAsync(). This method can run a task asynchronously and doesn't return a result. In other words, it takes a Runnable object and returns CompletableFuture<Void>; this is shown in the following code:

public static void printOrder() {  
  
 CompletableFuture<Void> cfPrintOrder   
 = CompletableFuture.runAsync(new Runnable() {  
  
 @Override  
 public void run() {  
 logger.info(() -> "Order is printed by: "  
 + Thread.currentThread().getName());  
 Thread.sleep(500);  
 }  
 });  
  
 cfPrintOrder.get(); // block until the order is printed  
 logger.info("Customer order was printed ...\n");  
}Copy

Or, we can write it using a lambda:

public static void printOrder() {  
  
 CompletableFuture<Void> cfPrintOrder   
 = CompletableFuture.runAsync(() -> {  
  
 logger.info(() -> "Order is printed by: "  
 + Thread.currentThread().getName());  
 Thread.sleep(500);  
 });  
  
 cfPrintOrder.get(); // block until the order is printed  
 logger.info("Customer order was printed ...\n");  
}Copy

Running an asynchronous task and returning a result

User problem: *Fetch the order summary of a certain customer.*

This time, the asynchronous task must return a result, and so runAsync() is not useful. This is a job for supplyAsync(). It takes Supplier<T> and returns CompletableFuture<T>. T is the type of the result obtained from this supplier via the get() method. In code lines, we can solve this problem as follows:

public static void fetchOrderSummary() {  
  
 CompletableFuture<String> cfOrderSummary   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Fetch order summary by: "  
 + Thread.currentThread().getName());  
 Thread.sleep(500);  
  
 return "Order Summary #93443";  
 });  
  
 // wait for summary to be available, this is blocking  
 String summary = cfOrderSummary.get();  
 logger.info(() -> "Order summary: " + summary + "\n");  
}Copy

Running an asynchronous task and returning a result via an explicit thread pool

User problem: *Fetch the order summary of a certain customer.*

By default, as in the preceding examples, the asynchronous tasks are executed in threads obtained from the global ForkJoinPool.commonPool(). By simply logging Thread.currentThread().getName(), we see something as ForkJoinPool.commonPool-worker-3.

But we can also use an explicit Executor custom thread pool. All the CompletableFuture methods that are capable of running asynchronous tasks provide a flavor that takes Executor.

Here is an example of using a single thread pool:

public static void fetchOrderSummaryExecutor() {  
  
 ExecutorService executor = Executors.newSingleThreadExecutor();  
  
 CompletableFuture<String> cfOrderSummary   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Fetch order summary by: "  
 + Thread.currentThread().getName());  
 Thread.sleep(500);  
  
 return "Order Summary #91022";  
 }, executor);  
  
 // wait for summary to be available, this is blocking  
 String summary = cfOrderSummary.get();  
 logger.info(() -> "Order summary: " + summary + "\n");  
 executor.shutdownNow();  
}Copy

Attaching a callback that processes the result of an asynchronous task and returns a result

User problem: *Fetch the order invoice of a certain customer and, afterward, compute the total and sign it.*

Relying on blocking get() is not very useful for such problems. What we need is a callback method that will be automatically called when the result of CompletableFuture is available.

So, we don't want to wait for the result. When the invoice is ready (this is the result of CompletableFuture), a callback method should compute the total value, and, afterward, another callback should sign it. This can be achieved via the thenApply() method.

The thenApply() method is useful for processing and transforming the result of CompletableFuture when it arrives. It takes Function<T, R> as an argument. Let's see it at work:

public static void fetchInvoiceTotalSign() {  
  
 CompletableFuture<String> cfFetchInvoice   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Fetch invoice by: "  
 + Thread.currentThread().getName());  
 Thread.sleep(500);  
  
 return "Invoice #3344";  
 });  
  
 CompletableFuture<String> cfTotalSign = cfFetchInvoice  
 .thenApply(o -> o + " Total: $145")  
 .thenApply(o -> o + " Signed");  
  
 String result = cfTotalSign.get();  
 logger.info(() -> "Invoice: " + result + "\n");  
}Copy

Or, we can chain it as follows:

public static void fetchInvoiceTotalSign() {  
  
 CompletableFuture<String> cfTotalSign   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Fetch invoice by: "  
 + Thread.currentThread().getName());  
 Thread.sleep(500);  
  
 return "Invoice #3344";  
 }).thenApply(o -> o + " Total: $145")  
 .thenApply(o -> o + " Signed");  
  
 String result = cfTotalSign.get();  
 logger.info(() -> "Invoice: " + result + "\n");  
}Copy

Check also applyToEither() and applyToEitherAsync(). When either this or the other given stage completes in a normal way, these two methods return a new completion stage that is executed with the result as an argument to the supplied function.

Attaching a callback that processes the result of an asynchronous task and returns void

User problem: *Fetch the order of a certain customer and print it.*

Typically, a callback that doesn't return a result acts as a terminal action of an asynchronous pipeline.

This behavior can be obtained via the thenAccept() method. It takes Consumer<T> and returns CompletableFuture<Void>. This method can process and transform the result of CompletableFuture, but doesn't return a result. So, it can take an order, which is the result of CompletableFuture, and print it as shown in the following snippet of code:

public static void fetchAndPrintOrder() {  
  
 CompletableFuture<String> cfFetchOrder   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Fetch order by: "  
 + Thread.currentThread().getName());  
 Thread.sleep(500);  
  
 return "Order #1024";  
 });  
  
 CompletableFuture<Void> cfPrintOrder = cfFetchOrder.thenAccept(  
 o -> logger.info(() -> "Printing order " + o +  
 " by: " + Thread.currentThread().getName()));  
  
 cfPrintOrder.get();  
 logger.info("Order was fetched and printed \n");  
}Copy

Or, it can be more compact as follows:

public static void fetchAndPrintOrder() {  
  
 CompletableFuture<Void> cfFetchAndPrintOrder   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Fetch order by: "  
 + Thread.currentThread().getName());  
 Thread.sleep(500);  
  
 return "Order #1024";  
 }).thenAccept(  
 o -> logger.info(() -> "Printing order " + o + " by: "  
 + Thread.currentThread().getName()));  
  
 cfFetchAndPrintOrder.get();  
 logger.info("Order was fetched and printed \n");  
}Copy

Check also acceptEither() and acceptEitherAsync().

Attaching a callback that runs after an asynchronous task and returns void

User problem: *Deliver an order and notify the customer.*

Notifying the customer should be accomplished after delivering the order. This is just an SMS of the *Dear customer, your order has been delivered today*type, so the notification task doesn't need to know anything about the order. These kinds of tasks can be accomplished by thenRun(). This method takes Runnable and returns CompletableFuture<Void>. Let's see it at work:

public static void deliverOrderNotifyCustomer() {  
  
 CompletableFuture<Void> cfDeliverOrder   
 = CompletableFuture.runAsync(() -> {  
  
 logger.info(() -> "Order was delivered by: "  
 + Thread.currentThread().getName());  
 Thread.sleep(500);  
 });  
  
 CompletableFuture<Void> cfNotifyCustomer   
 = cfDeliverOrder.thenRun(() -> logger.info(  
 () -> "Dear customer, your order has been delivered today by:"  
 + Thread.currentThread().getName()));  
  
 cfNotifyCustomer.get();  
 logger.info(() -> "Order was delivered   
 and customer was notified \n");  
}Copy

For further parallelization, thenApply(), thenAccept(), and thenRun() are accompanied by thenApplyAsync(), thenAcceptAsync(), and thenRunAsync(). Each of these can rely on the global ForkJoinPool.commonPool() or a custom thread pool (Executor). While thenApply/Accept/Run() are executed in the same thread as the CompletableFuture task was executed before (or in the main thread), thenApplyAsync/AcceptAsync/RunAsync() may be executed in a different thread (from ForkJoinPool.commonPool() or a custom thread pool (Executor)).

Handling exceptions of an asynchronous task via exceptionally()

User problem: *Compute the total of an order. If something goes wrong, then throw an* IllegalStateException*.*

The following screenshots exemplify how exceptions are propagated in an asynchronous pipeline; the code in rectangles is not executed when an exception occurs at the point:

Texto, Carta

Descripción generada automáticamente

The following screenshot shows the exceptions in thenApply() and thenAccept():

Texto

Descripción generada automáticamente con confianza media

So, in supplyAsync(), if an exception occurs, then none of the following callbacks will be called. Moreover, the future will be resolved with this exception. The same rule applies for each callback. If the exception occurs in the first thenApply(), then the following thenApply() and thenAccept() will not be called.

If our attempt to computing the total of order ends up in an IllegalStateException, then we can rely on the exceptionally() callback which gives us a chance to recover. This method takes a Function<Throwable,​? extends T> and returns a CompletionStage<T>, therefore, a CompletableFuture. Let's see it at work:

public static void fetchOrderTotalException() {  
  
 CompletableFuture<Integer> cfTotalOrder   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Compute total: "  
 + Thread.currentThread().getName());  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Invoice service is not responding");  
 }  
  
 return 1000;  
 }).exceptionally(ex -> {  
 logger.severe(() -> "Exception: " + ex  
 + " Thread: " + Thread.currentThread().getName());  
  
 return 0;  
 });  
  
 int result = cfTotalOrder.get();  
 logger.info(() -> "Total: " + result + "\n");  
}Copy

In case of exception, the output will be as follows:

Compute total: ForkJoinPool.commonPool-worker-3  
Exception: java.lang.IllegalStateException: Invoice service  
 is not responding Thread: ForkJoinPool.commonPool-worker-3  
Total: 0Copy

Let's take a look at another problem.

User problem: *Fetch an invoice, compute the total, and sign. If something goes wrong then throw*IllegalStateException *and stop the process.*

If we fetch the invoice using supplyAsync(), compute the total using thenApply() and sign using another thenApply(), then we may think that the right implementation is as follows:

public static void fetchInvoiceTotalSignChainOfException()  
throws InterruptedException, ExecutionException {  
  
 CompletableFuture<String> cfFetchInvoice   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Fetch invoice by: "  
 + Thread.currentThread().getName());  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Invoice service is not responding");  
 }  
  
 return "Invoice #3344";  
 }).exceptionally(ex -> {  
 logger.severe(() -> "Exception: " + ex  
 + " Thread: " + Thread.currentThread().getName());  
  
 return "[Invoice-Exception]";  
 }).thenApply(o -> {  
 logger.info(() -> "Compute total by: "  
 + Thread.currentThread().getName());  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Total service is not responding");  
 }  
  
 return o + " Total: $145";  
 }).exceptionally(ex -> {  
 logger.severe(() -> "Exception: " + ex  
 + " Thread: " + Thread.currentThread().getName());  
  
 return "[Total-Exception]";  
 }).thenApply(o -> {  
 logger.info(() -> "Sign invoice by: "  
 + Thread.currentThread().getName());  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Signing service is not responding");  
 }  
  
 return o + " Signed";  
 }).exceptionally(ex -> {  
 logger.severe(() -> "Exception: " + ex  
 + " Thread: " + Thread.currentThread().getName());  
  
 return "[Sign-Exception]";  
 });  
  
 String result = cfFetchInvoice.get();  
 logger.info(() -> "Result: " + result + "\n");  
}Copy

Well, the issue here is that we may face an output as follows:

[INFO] Fetch invoice by: ForkJoinPool.commonPool-worker-3  
[SEVERE] Exception: java.lang.IllegalStateException: Invoice service  
 is not responding Thread: ForkJoinPool.commonPool-worker-3  
[INFO] Compute total by: ForkJoinPool.commonPool-worker-3  
[INFO] Sign invoice by: ForkJoinPool.commonPool-worker-3  
[SEVERE] Exception: java.lang.IllegalStateException: Signing service  
 is not responding Thread: ForkJoinPool.commonPool-worker-3  
[INFO] Result: [Sign-Exception]Copy

Even if the invoice couldn't be fetched, we would continue to compute the total and sign it. Obviously, this doesn't make sense. If the invoice cannot be fetched, or the total cannot be computed, then we expect to abort the process. While this implementation can be a good fit when we can recover and continue, it is definitely no good for our scenario. For our scenario, the following implementation is needed:

public static void fetchInvoiceTotalSignException()  
throws InterruptedException, ExecutionException {  
  
 CompletableFuture<String> cfFetchInvoice   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Fetch invoice by: "  
 + Thread.currentThread().getName());  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Invoice service is not responding");  
 }  
  
 return "Invoice #3344";  
 }).thenApply(o -> {  
 logger.info(() -> "Compute total by: "  
 + Thread.currentThread().getName());  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Total service is not responding");  
 }  
  
 return o + " Total: $145";  
 }).thenApply(o -> {  
 logger.info(() -> "Sign invoice by: "  
 + Thread.currentThread().getName());  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Signing service is not responding");  
 }  
  
 return o + " Signed";  
 }).exceptionally(ex -> {  
 logger.severe(() -> "Exception: " + ex  
 + " Thread: " + Thread.currentThread().getName());  
  
 return "[No-Invoice-Exception]";  
 });  
  
 String result = cfFetchInvoice.get();  
 logger.info(() -> "Result: " + result + "\n");  
}Copy

This time, an exception occurring in any of the implied CompletableFuture will stop the process. Here is a possible output:

[INFO ] Fetch invoice by: ForkJoinPool.commonPool-worker-3  
[SEVERE] Exception: java.lang.IllegalStateException: Invoice service  
 is not responding Thread: ForkJoinPool.commonPool-worker-3  
[INFO ] Result: [No-Invoice-Exception]Copy

Starting with JDK 12, the exceptional cases can be further parallelized via exceptionallyAsync() that can use the same thread as the code that caused the exception or a thread from the given thread pool (Executor). Here is an example:

public static void fetchOrderTotalExceptionAsync() {  
  
 ExecutorService executor = Executors.newSingleThreadExecutor();  
  
 CompletableFuture<Integer> totalOrder   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Compute total by: "  
 + Thread.currentThread().getName());  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Computing service is not responding");  
 }  
  
 return 1000;  
 }).exceptionallyAsync(ex -> {  
 logger.severe(() -> "Exception: " + ex   
 + " Thread: " + Thread.currentThread().getName());  
  
 return 0;  
 }, executor);  
  
 int result = totalOrder.get();  
 logger.info(() -> "Total: " + result + "\n");  
 executor.shutdownNow();  
}Copy

The output reveals that the code that caused the exception was executed by a thread named ForkJoinPool.commonPool-worker-3, while the exceptional code was executed by a thread from the given thread pool named pool-1-thread-1:

Compute total by: ForkJoinPool.commonPool-worker-3  
Exception: java.lang.IllegalStateException: Computing service is  
 not responding Thread: pool-1-thread-1  
Total: 0Copy

JDK 12 exceptionallyCompose()

User problem: *Fetch a printer IP via the printing service or fallback to the backup printer IP. Or, generally speaking, when this stage completes exceptionally, it should be composed using the results of the supplied function applied to this stage's exception*.

We have CompletableFuture that fetches an IP of a printer managed by the printing service. If the service is not responding then it throws an exception as follows:

CompletableFuture<String> cfServicePrinterIp   
 = CompletableFuture.supplyAsync(() -> {  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Printing service is not responding");  
 }  
  
 return "192.168.1.0";  
});Copy

We also have CompletableFuture that fetches the IP of the backup printer:

CompletableFuture<String> cfBackupPrinterIp   
 = CompletableFuture.supplyAsync(() -> {  
  
 return "192.192.192.192";  
});Copy

Now, if the printing service is not available, then we should rely on the backup printer. This can be accomplished via the JDK 12 exceptionallyCompose() as follows:

CompletableFuture<Void> printInvoice   
 = cfServicePrinterIp.exceptionallyCompose(th -> {  
  
 logger.severe(() -> "Exception: " + th  
 + " Thread: " + Thread.currentThread().getName());  
  
 return cfBackupPrinterIp;  
}).thenAccept((ip) -> logger.info(() -> "Printing at: " + ip));Copy

Calling printInvoice.get() may reveal one of the following results:

* If the printing service is available:

[INFO] Printing at: 192.168.1.0Copy

* If the printing service is not available:

[SEVERE] Exception: java.util.concurrent.CompletionException ...  
[INFO] Printing at: 192.192.192.192Copy

For further parallelization, we can rely on exceptionallyComposeAsync().

Handling exceptions of an asynchronous task via handle()

User problem: *Compute the total of an order. If something goes wrong then throw an*IllegalStateException*.*

Sometimes we want to execute an exceptional block of code even if an exception did not occur. Like the finally clause of a try-catch block. This is possible using the handle() callback. This method is called whether or not an exception occurred, and is somehow like a catch + finally. It takes a function used to compute the value of the returned CompletionStage, BiFunction<? super T,​Throwable,​? extends U> and returns CompletionStage<U> (U is the function's return type).

Let's see it at work:

public static void fetchOrderTotalHandle() {  
  
 CompletableFuture<Integer> totalOrder   
 = CompletableFuture.supplyAsync(() -> {  
  
 logger.info(() -> "Compute total by: "  
 + Thread.currentThread().getName());  
  
 int surrogate = new Random().nextInt(1000);  
 if (surrogate < 500) {  
 throw new IllegalStateException(  
 "Computing service is not responding");  
 }  
  
 return 1000;  
 }).handle((res, ex) -> {  
 if (ex != null) {  
 logger.severe(() -> "Exception: " + ex  
 + " Thread: " + Thread.currentThread().getName());  
  
 return 0;  
 }  
  
 if (res != null) {  
 int vat = res \* 24 / 100;  
 res += vat;  
 }  
  
 return res;  
 });  
  
 int result = totalOrder.get();  
 logger.info(() -> "Total: " + result + "\n");  
}Copy

Notice that res will be null; otherwise, the ex will be null if an exception occurs.

If we need to complete with an exception, then we can proceed via completeExceptionally() as in the following example:

CompletableFuture<Integer> cf = new CompletableFuture<>();  
...  
cf.completeExceptionally(new RuntimeException("Ops!"));  
...  
cf.get(); // ExecutionException : RuntimeExceptionCopy

Canceling the execution and throwing CancellationException can be done via the cancel() method:

CompletableFuture<Integer> cf = new CompletableFuture<>();  
...  
// is not important if the argument is set to true or false  
cf.cancel(true/false);  
...  
cf.get(); // CancellationExceptionCopy

Explicitly complete a CompletableFuture

A CompletableFuture can be explicitly completed using complete​(T value), completeAsync​(Supplier<? extends T> supplier), and completeAsync​(Supplier<? extends T> supplier, Executor executor). T is the value returned by get(). Here it is a method that creates CompletableFuture and returns it immediately. Another thread is responsible for executing some tax computations and completing the CompletableFuture with the corresponding result:

public static CompletableFuture<Integer> taxes() {  
  
 CompletableFuture<Integer> completableFuture   
 = new CompletableFuture<>();  
  
 new Thread(() -> {  
 int result = new Random().nextInt(100);  
 Thread.sleep(10);  
  
 completableFuture.complete(result);  
 }).start();  
  
 return completableFuture;  
}Copy

And, let's call this method:

logger.info("Computing taxes ...");  
  
CompletableFuture<Integer> cfTaxes = CustomerAsyncs.taxes();  
  
while (!cfTaxes.isDone()) {  
 logger.info("Still computing ...");  
}  
  
int result = cfTaxes.get();  
logger.info(() -> "Result: " + result);Copy

A possible output will be the following:

[14:09:40] [INFO ] Computing taxes ...  
[14:09:40] [INFO ] Still computing ...  
[14:09:40] [INFO ] Still computing ...  
...  
[14:09:40] [INFO ] Still computing ...  
[14:09:40] [INFO ] Result: 17Copy

If we already know the result of CompletableFuture, then we can call completedFuture​(U value) as in the following example:

CompletableFuture<String> completableFuture   
 = CompletableFuture.completedFuture("How are you?");  
  
String result = completableFuture.get();  
logger.info(() -> "Result: " + result); // Result: How are you?Copy

Also, check the documentation of whenComplete() and whenCompleteAsync().

217. Combining multiple CompletableFuture instances

In most cases, combining CompletableFuture instances can be accomplished using the following:

* thenCompose()
* thenCombine()
* allOf()
* anyOf()

By combining CompletableFuture instances, we can shape complex asynchronous solutions. This way, multiple CompletableFuture instances can combine their powers for reaching a common goal.

Combining via thenCompose()

Let's assume that we have the following two CompletableFuture instances in a helper class named CustomerAsyncs:

private static CompletableFuture<String>   
 fetchOrder(String customerId) {  
  
 return CompletableFuture.supplyAsync(() -> {  
 return "Order of " + customerId;  
 });  
}  
  
private static CompletableFuture<Integer> computeTotal(String order) {  
  
 return CompletableFuture.supplyAsync(() -> {  
 return order.length() + new Random().nextInt(1000);  
 });  
}Copy

Now, we want to fetch the order of a certain customer, and, once the order is available, we want to compute the total of this order. This means that we need to call fetchOrder() and afterward computeTotal(). We can do this via thenApply():

CompletableFuture<CompletableFuture<Integer>> cfTotal   
 = fetchOrder(customerId).thenApply(o -> computeTotal(o));  
  
int total = cfTotal.get().get();Copy

Obviously, this is not a convenient solution since the result is of the CompletableFuture<CompletableFuture<Integer>> type. In order to avoid the nesting of CompletableFuture instances, we can rely on thenCompose() as follows:

CompletableFuture<Integer> cfTotal   
 = fetchOrder(customerId).thenCompose(o -> computeTotal(o));  
  
int total = cfTotal.get();  
  
// e.g., Total: 734  
logger.info(() -> "Total: " + total);Copy

Whenever we need to obtain a flattened result from a chain of CompletableFuture instances, we can use thenCompose(). This way we avoid nested examples of CompletableFuture instances.

Further parallelization can be obtained using thenComposeAsync().

Combining via thenCombine()

While thenCompose() is useful to chain two dependent CompletableFuture instances, thenCombine() is useful to chain two independent instances of CompletableFuture. When both CompletableFuture instances complete we can continue .

Let's assume that we have the following two CompletableFuture instances:

private static CompletableFuture<Integer> computeTotal(String order) {  
  
 return CompletableFuture.supplyAsync(() -> {  
 return order.length() + new Random().nextInt(1000);  
 });  
}  
  
private static CompletableFuture<String> packProducts(String order) {  
  
 return CompletableFuture.supplyAsync(() -> {  
 return "Order: " + order   
 + " | Product 1, Product 2, Product 3, ... ";  
 });  
}Copy

In order to deliver a customer order, we need to compute the total (for emitting the invoice) and pack the ordered products. These two actions can be accomplished in parallel. In the end, we deliver the parcel containing the ordered products and the invoice. Achieving this via thenCombine() can be done as follows:

CompletableFuture<String> cfParcel = computeTotal(order)  
 .thenCombine(packProducts(order), (total, products) -> {  
 return "Parcel-[" + products + " Invoice: $" + total + "]";  
 });  
  
String parcel = cfParcel.get();  
  
// e.g. Delivering: Parcel-[Order: #332 | Product 1, Product 2,  
// Product 3, ... Invoice: $314]  
logger.info(() -> "Delivering: " + parcel);Copy

The callback function given to thenCombine() will be invoked after both CompletableFuture instances are complete.

If all we need is to do something when two CompletableFuture instances complete normally (this and another one) then we can rely on thenAcceptBoth(). This method returns a new CompletableFuture that is executed with the two results as arguments to the supplied action. The two results are this and the other given stage (they must complete normally). Here is an example:

CompletableFuture<Void> voidResult = CompletableFuture  
 .supplyAsync(() -> "Pick")  
 .thenAcceptBoth(CompletableFuture.supplyAsync(() -> " me"),  
 (pick, me) -> System.out.println(pick + me));Copy

If the results of these two CompletableFuture instances are not needed, then runAfterBoth() is much preferred.

Combining via allOf()

Let's assume that we want to download the following list of invoices:

List<String> invoices = Arrays.asList("#2334", "#122", "#55");Copy

This can be seen as a bunch of independent tasks that can be accomplished in parallel, so we can do it using CompletableFuture as follows:

public static CompletableFuture<String>   
 downloadInvoices(String invoice) {  
  
 return CompletableFuture.supplyAsync(() -> {  
 logger.info(() -> "Downloading invoice: " + invoice);  
  
 return "Downloaded invoice: " + invoice;  
 });  
}  
  
CompletableFuture<String> [] cfInvoices = invoices.stream()  
 .map(CustomerAsyncs::downloadInvoices)  
 .toArray(CompletableFuture[]::new);Copy

At this point, we have an array of CompletableFuture instances, and, therefore, an array of asynchronous computations. Furthermore, we want to run all of them in parallel. This can be accomplished using the allOf​(CompletableFuture<?>... cfs) method. The result consists of a CompletableFuture<Void> as follows:

CompletableFuture<Void> cfDownloaded   
 = CompletableFuture.allOf(cfInvoices);  
cfDownloaded.get();Copy

Obviously, the result of allOf() is not very useful. What can we do with CompletableFuture<Void>? There are definitely many problems when we need the results of each computation involved in this parallelization, so we need a solution for fetching the results instead of relying on CompletableFuture<Void>.

We can solve this problem via thenApply() as follows:

List<String> results = cfDownloaded.thenApply(e -> {  
 List<String> downloaded = new ArrayList<>();  
  
 for (CompletableFuture<String> cfInvoice: cfInvoices) {  
 downloaded.add(cfInvoice.join());  
 }  
  
 return downloaded;  
}).get();Copy

The join() method is similar to get(), but, if the underlying CompletableFuture completes exceptionally, it throws an unchecked exception .

Since we are calling join() after all the involved CompletableFuture have completed, there is no blocking point.

The returned List<String> contains the results obtained by calling the downloadInvoices() method as follows:

Downloaded invoice: #2334  
  
Downloaded invoice: #122  
  
Downloaded invoice: #55Copy

Combining via anyOf()

Let's assume that we want to organize a raffle for our customers:

List<String> customers = Arrays.asList(  
 "#1", "#4", "#2", "#7", "#6", "#5"  
);Copy

We can start to solve this problem by defining the following trivial method:

public static CompletableFuture<String> raffle(String customerId) {  
  
 return CompletableFuture.supplyAsync(() -> {  
 Thread.sleep(new Random().nextInt(5000));  
  
 return customerId;  
 });  
}Copy

Now, we can create an array of CompletableFuture<String> instances, as follows:

CompletableFuture<String>[] cfCustomers = customers.stream()  
 .map(CustomerAsyncs::raffle)  
 .toArray(CompletableFuture[]::new);Copy

To find the winner of the raffle, we want to run cfCustomers in parallel, and the first CompletableFuture that completes is the winner. Since the raffle() method blocks for a random number of seconds, the winner will be randomly chosen. We are not interested in the rest of the CompletableFuture instances, so they should be completed immediately after the winner has been chosen.

This is a job for anyOf​(CompletableFuture<?>... cfs). It returns a new CompletableFuture that is completed when any of the involved CompletableFuture instances completes. Let's see it at work:

CompletableFuture<Object> cfWinner   
 = CompletableFuture.anyOf(cfCustomers);  
  
Object winner = cfWinner.get();  
  
// e.g., Winner: #2  
logger.info(() -> "Winner: " + winner);Copy

Pay attention to scenarios that rely on CompletableFuture that return results of different types. Since anyOf() returns CompletableFuture<Object>, it is difficult to know the CompletableFuture types that have completed first.

218. Optimizing busy waiting

The *busy waiting* technique (also known as *busy-looping* or *spinning*) consists of a loop that checks a condition (typically, a flag condition). For example, the following loop waits for a service to start:

private volatile boolean serviceAvailable;  
...  
while (!serviceAvailable) {}Copy

Java 9 introduced the Thread.onSpinWait() method. This is a hotspot that gives the JVM a hint that the following code is in a spin loop:

while (!serviceAvailable) {  
 Thread.onSpinWait();  
}Copy

Intel SSE2 PAUSE instruction is provided precisely for this reason. For more details, see the Intel official documentation. Also have a look at this link: <https://software.intel.com/en-us/articles/benefitting-power-and-performance-sleep-loops>.

If we add this while loop in a context, then we obtain the following class:

public class StartService implements Runnable {  
  
 private volatile boolean serviceAvailable;  
  
 @Override  
 public void run() {  
 System.out.println("Wait for service to be available ...");  
  
 while (!serviceAvailable) {  
 // Use a spin-wait hint (ask the processor to  
 // optimize the resource)  
 // This should perform better if the underlying  
 // hardware supports the hint  
 Thread.onSpinWait();  
 }  
  
 serviceRun();  
 }  
  
 public void serviceRun() {  
 System.out.println("Service is running ...");  
 }  
  
 public void setServiceAvailable(boolean serviceAvailable) {  
 this.serviceAvailable = serviceAvailable;  
 }  
}Copy

And, we can easily test it (do not expect to see the effect of onSpinWait()):

StartService startService = new StartService();  
new Thread(startService).start();  
  
Thread.sleep(5000);  
  
startService.setServiceAvailable(true);Copy

219. Task Cancellation

Cancellation is a common technique used for forcibly stopping or completing a task that is currently running. A canceled task will not complete naturally. Cancellation should have no effect on an already completed task. Think of it as a **Cancel** button of a GUI.

Java doesn't provide a preemptive way for stopping a thread. Therefore for canceling a task, a common practice is to rely on a loop that uses a flag condition. The task responsibility is to check this flag periodically, and when it finds the flag set, then it should stop as fast as possible. The following code is an example of this:

public class RandomList implements Runnable {  
 private volatile boolean cancelled;  
 private final List<Integer> randoms = new CopyOnWriteArrayList<>();  
 private final Random rnd = new Random();  
  
 @Override  
 public void run() {  
 while (!cancelled) {  
 randoms.add(rnd.nextInt(100));  
 }  
 }  
  
 public void cancel() {  
 cancelled = true;  
 }  
  
 public List<Integer> getRandoms() {  
 return randoms;  
 }  
}Copy

The focus here is on the canceled variable. Notice that this variable was declared as volatile (also known as the lighter-weight synchronization mechanism). Being a volatile variable, it is not cached by threads and operations on it are not reordered in memory; therefore, a thread cannot see an old value. Any thread that reads a volatile field will see the most recently written value. This is exactly what we need in order to communicate the cancellation action to all running threads that are interested in this action. The following diagram depicts how volatile and non-volatile work:

Diagrama

Descripción generada automáticamente

Notice that the volatile variables are not a good fit for read-modify-write scenarios. For such scenarios, we will rely on atomic variables (for example, AtomicBoolean, AtomicInteger, AtomicReference, and so on).

Now, let's provide a simple snippet of code for canceling the task implemented in RandomList:

RandomList rl = new RandomList();  
  
ExecutorService executor = Executors.newFixedThreadPool(10);  
  
for (int i = 0; i < 100; i++) {  
 executor.execute(rl);  
}  
  
Thread.sleep(100);  
  
rl.cancel();  
  
System.out.println(rl.getRandoms());Copy

220. ThreadLocal

Java threads share the same memory, but sometimes we need to have dedicated memory for each thread. Java provides ThreadLocal as an approach for storing and retrieving values for each thread separately. A single instance of ThreadLocal can store and retrieve values of multiple threads. If thread A stores the x value and thread B stores the y value in the same instance of ThreadLocal then, later on, thread A retrieves the x value and thread B retrieves the y value.

Java ThreadLocal is typically used in the following two scenarios:



For providing per-thread instances (thread-safety and memory efficiency)



For providing per-thread context

Let's take a look at problems for each scenario in the next sections.

Per-thread instances

Let's assume that we have a single-thread application that uses a global variable of the StringBuilder type. In order to transform the application in a multithreaded application, we have to deal with StringBuilder, which is not thread-safe. Basically, we have several approaches such as synchronization and StringBuffer or other approaches. However, we can use ThreadLocal as well. The main idea here is to provide a separate StringBuilder to each thread. Using ThreadLocal, we can do it as follows:

private static final ThreadLocal<StringBuilder>   
 threadLocal = new ThreadLocal<>() {  
  
 @Override  
 protected StringBuilder initialValue() {  
 return new StringBuilder("ThreadSafe ");  
 }  
};Copy

The current thread's *initial value* for this thread-local variable is set via the initialValue() method. In Java 8, this can be re-written via withInitial() as follows:

private static final ThreadLocal<StringBuilder> threadLocal   
 = ThreadLocal.<StringBuilder> withInitial(() -> {  
  
 return new StringBuilder("Thread-safe ");  
});Copy

Working with ThreadLocal is done using get() and set(). Every call of set() stores the given value in a memory region that only the current thread has access to. Later on, calling get() will retrieve the value from this region. In addition, once the job is done, it is advisable to avoid memory leaks by calling the remove() or set(null) methods on the ThreadLocal instance.

Let's see a ThreadLocal at work using a Runnable:

public class ThreadSafeStringBuilder implements Runnable {  
  
 private static final Logger logger =  
 Logger.getLogger(ThreadSafeStringBuilder.class.getName());  
 private static final Random rnd = new Random();  
  
 private static final ThreadLocal<StringBuilder> threadLocal   
 = ThreadLocal.<StringBuilder> withInitial(() -> {  
  
 return new StringBuilder("Thread-safe ");  
 });  
  
 @Override  
 public void run() {  
 logger.info(() -> "-> " + Thread.currentThread().getName()   
 + " [" + threadLocal.get() + "]");  
  
 Thread.sleep(rnd.nextInt(2000));  
  
 // threadLocal.set(new StringBuilder(  
 // Thread.currentThread().getName()));  
 threadLocal.get().append(Thread.currentThread().getName());  
  
 logger.info(() -> "-> " + Thread.currentThread().getName()   
 + " [" + threadLocal.get() + "]");  
  
 threadLocal.set(null);  
 // threadLocal.remove();  
  
 logger.info(() -> "-> " + Thread.currentThread().getName()   
 + " [" + threadLocal.get() + "]");  
 }  
}Copy

And, let's test it using several threads:

ThreadSafeStringBuilder threadSafe = new ThreadSafeStringBuilder();  
  
for (int i = 0; i < 3; i++) {  
 new Thread(threadSafe, "thread-" + i).start();  
}Copy

The output reveals that each thread accesses its own StringBuilder:

[14:26:39] [INFO] -> thread-1 [Thread-safe ]  
[14:26:39] [INFO] -> thread-0 [Thread-safe ]  
[14:26:39] [INFO] -> thread-2 [Thread-safe ]  
[14:26:40] [INFO] -> thread-0 [Thread-safe thread-0]  
[14:26:40] [INFO] -> thread-0 [null]  
[14:26:41] [INFO] -> thread-1 [Thread-safe thread-1]  
[14:26:41] [INFO] -> thread-1 [null]  
[14:26:41] [INFO] -> thread-2 [Thread-safe thread-2]  
[14:26:41] [INFO] -> thread-2 [null]Copy

In scenarios such as the preceding one, ExecutorService can be used as well.

Here is another snippet of code that provides a JDBC Connection to each thread:

private static final ThreadLocal<Connection> connections   
 = ThreadLocal.<Connection> withInitial(() -> {  
  
 try {  
 return DriverManager.getConnection("jdbc:mysql://...");  
 } catch (SQLException ex) {  
 throw new RuntimeException("Connection acquisition failed!", ex);  
 }  
});  
  
public static Connection getConnection() {  
 return connections.get();  
}Copy

Per-thread context

Let's assume that we have the following Order class:

public class Order {  
  
 private final int customerId;  
  
 public Order(int customerId) {  
 this.customerId = customerId;  
 }  
  
 // getter and toString() omitted for brevity  
}Copy

And, we write CustomerOrder as follows:

public class CustomerOrder implements Runnable {  
  
 private static final Logger logger  
 = Logger.getLogger(CustomerOrder.class.getName());  
 private static final Random rnd = new Random();  
  
 private static final ThreadLocal<Order>   
 customerOrder = new ThreadLocal<>();  
  
 private final int customerId;  
  
 public CustomerOrder(int customerId) {  
 this.customerId = customerId;  
 }  
  
 @Override  
 public void run() {  
 logger.info(() -> "Given customer id: " + customerId   
 + " | " + customerOrder.get()   
 + " | " + Thread.currentThread().getName());  
  
 customerOrder.set(new Order(customerId));  
  
 try {  
 Thread.sleep(rnd.nextInt(2000));  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 }  
  
 logger.info(() -> "Given customer id: " + customerId   
 + " | " + customerOrder.get()   
 + " | " + Thread.currentThread().getName());  
  
 customerOrder.remove();  
 }  
}Copy

For each customerId, we have a dedicated thread that we control:

CustomerOrder co1 = new CustomerOrder(1);  
CustomerOrder co2 = new CustomerOrder(2);  
CustomerOrder co3 = new CustomerOrder(3);  
  
new Thread(co1).start();  
new Thread(co2).start();  
new Thread(co3).start();Copy

So, each thread modifies a certain instance of CustomerOrder (there is a particular thread for each instance).

The run() method fetches the order for the given customerId and stores it in the ThreadLocal variable, using the set() method.

A possible output will be as follows:

[14:48:20] [INFO]   
 Given customer id: 3 | null | Thread-2  
[14:48:20] [INFO]   
 Given customer id: 2 | null | Thread-1  
[14:48:20] [INFO]   
 Given customer id: 1 | null | Thread-0  
  
[14:48:20] [INFO]   
 Given customer id: 2 | Order{customerId=2} | Thread-1  
[14:48:21] [INFO]   
 Given customer id: 3 | Order{customerId=3} | Thread-2  
[14:48:21] [INFO]   
 Given customer id: 1 | Order{customerId=1} | Thread-0Copy

In scenarios like the preceding one, avoid using ExecutorService. There is no guarantee that each Runnable (of a given customerId) will be handled by the same thread at every execution. This may lead to weird results.

221. Atomic variables

A naive approach for counting all numbers from 1 to 1,000,000 via Runnable may look as follows:

public class Incrementator implements Runnable {  
  
 public [static] int count = 0;  
  
 @Override  
 public void run() {  
 count++;  
 }  
  
 public int getCount() {  
 return count;  
 }  
}Copy

And, let's spin-up five threads that will increment the count variable concurrently:

Incrementator nonAtomicInc = new Incrementator();  
ExecutorService executor = Executors.newFixedThreadPool(5);  
  
for (int i = 0; i < 1 \_000\_000; i++) {  
 executor.execute(nonAtomicInc);  
}Copy

But, if we run this code several times, we get different results as follows:

997776, 997122, 997681 ...Copy

So, why don't we get the expected result, 1,000,000? The reason is because count++ is not an atomic operation/action. It consists of three atomic bytecode instructions:

iload\_1  
iinc 1, 1  
istore\_1Copy

During one thread, read the count value and increment it by one, and another thread reads the older value leading to a wrong result. In a multi-threading application, the scheduler can halt the execution of the current thread between each of these bytecode instructions and start a new thread, which works on the same variable. We can fix things via synchronization or, even better, via atomic variables.

Atomic variable classes are available in java.util.concurrent.atomic. They are wrapper classes that limit the scope of contention to a single variable; they are much more lightweight than Java synchronization and are based on **CAS** (short for **Compare and Swap**: modern CPUs support this technique in which it compares the content of a given memory location with a given value and updates it to a new value if the current value equals the expected value). Mainly, these are atomic compound actions that affect a single value in a lock-free manner similar to volatile. The most used atomic variables are the scalars:

* AtomicInteger
* AtomicLong
* AtomicBoolean
* AtomicReference

And, the following are for arrays:

* AtomicIntegerArray
* AtomicLongArray
* AtomicReferenceArray

Let's rewrite our example via AtomicInteger:

public class AtomicIncrementator implements Runnable {  
  
 public static AtomicInteger count = new AtomicInteger();  
  
 @Override  
 public void run() {  
 count.incrementAndGet();  
 }  
  
 public int getCount() {  
 return count.get();  
 }  
}Copy

Notice that, instead of count++, we wrote count.incrementAndGet(). This is just one of the methods provided by AtomicInteger. This method atomically increments the variable and returns the new value. This time, the count will be 1,000,000.

The following table contains several methods of AtomicInteger that are commonly used. The left column contains the methods, while the right column contains the non-atomic meaning:

AtomicInteger ai = new AtomicInteger(0); // atomic  
int i = 0; // non-atomic  
  
// and  
int q = 5;  
int r;  
  
// and  
int e = 0;  
boolean b;Copy

|  |  |
| --- | --- |
| **Atomic operation** | **Non-atomic counterpart** |
| r = ai.get(); | r = i; |
| ai.set(q); | i = q; |
| r = ai.incrementAndGet(); | r = ++i; |
| r = ai.getAndIncrement(); | r = i++; |
| r = ai.decrementAndGet(); | r = --i; |
| r = ai.getAndDecrement(); | r = i--; |
| r = ai.addAndGet(q); | i = i + q; r = i; |
| r = ai.getAndAdd(q); | r = i; i = i + q; |
| r = ai.getAndSet(q); | r = i; i = q; |
| b = ai.compareAndSet(e, q); | if (i == e) { i = q; return true; } else { return false; } |

Let's tackle several problems via atomic operations:



Update the elements of an array via updateAndGet​(IntUnaryOperator updateFunction):

// [9, 16, 4, 25]  
AtomicIntegerArray atomicArray  
 = new AtomicIntegerArray(new int[] {3, 4, 2, 5});  
  
for (int i = 0; i < atomicArray.length(); i++) {  
 atomicArray.updateAndGet(i, elem -> elem \* elem);  
}Copy



Update a single integer via updateAndGet​(IntUnaryOperator updateFunction):

// 15  
AtomicInteger nr = new AtomicInteger(3);  
int result = nr.updateAndGet(x -> 5 \* x);Copy



Update a single integer via accumulateAndGet​(int x, IntBinaryOperator accumulatorFunction):

// 15  
AtomicInteger nr = new AtomicInteger(3);  
// x = 3, y = 5  
int result = nr.accumulateAndGet(5, (x, y) -> x \* y);Copy



Update a single integer via addAndGet​(int delta):

// 7  
AtomicInteger nr = new AtomicInteger(3);  
int result = nr.addAndGet(4);Copy



Update a single integer via compareAndSet​(int expectedValue, int newValue):

// 5, true  
AtomicInteger nr = new AtomicInteger(3);  
boolean wasSet = nr.compareAndSet(3, 5);Copy

Starting with JDK 9, atomic variable classes have been enriched with several methods such as get/setPlain(), get/setOpaque(), getAcquire(), and their companions. To gain an understanding of these methods, have a look at *Using JDK 9 Memory Order Modes* by Doug Lea, available at [http://gee.cs.oswego.edu/dl/html/j9mm.html,](http://gee.cs.oswego.edu/dl/html/j9mm.html) at the time of writing.

Adders and accumulators

Following the Java API documentation, in cases of multithreading applications that update frequently but read less frequently, it is recommended to rely on LongAdder, DoubleAdder, LongAccumulator, and DoubleAccumulator, instead of the Atomic*Foo* classes. For such scenarios, these classes are designed to optimize the usage of threads.

This means that, instead of using AtomicInteger for counting the integers from 1 to 1,000,000, we can use LongAdder as follows:

public class AtomicAdder implements Runnable {  
  
 public static LongAdder count = new LongAdder();  
  
 @Override  
 public void run() {  
  
 count.add(1);  
 }  
  
 public long getCount() {  
  
 return count.sum();  
 }  
}Copy

Alternatively, we can use LongAccumulator as follows:

public class AtomicAccumulator implements Runnable {  
  
 public static LongAccumulator count  
 = new LongAccumulator(Long::sum, 0);  
  
 @Override  
 public void run() {  
  
 count.accumulate(1);  
 }  
  
 public long getCount() {  
  
 return count.get();  
 }  
}Copy

The LongAdder and DoubleAdder are right for scenarios that imply additions (operations specific to additions), while LongAccumulator and DoubleAccumulator are right for scenarios that rely on a given function to combine values.

222. ReentrantLock

The Lock interface contains a set of locking operations that can be explicitly used to fine-tune the locking process (it provides more control than intrinsic locking). Among them, we have polled, unconditional, timed, and interruptible lock acquisition. Basically, Lock exposes the futures of the synchronized keyword with additional capabilities. The Lock interface is shown in the following code:

public interface Lock {  
 void lock();  
 void lockInterruptibly() throws InterruptedException;  
 boolean tryLock();  
 boolean tryLock(long timeout, TimeUnit unit)  
 throws InterruptedException;  
 void unlock();  
 Condition newCondition();  
}Copy

One of the implementations of Lock is ReentrantLock. A *reentrant* lock acts as follows: when the thread enters for the first time into the lock, a hold count is set to one. Before unlocking, the thread can re-enter the lock causing the hold count to be incremented by one for each entry. Each unlock request decrements the hold count by one, and, when the hold count is zero, the locked resource is opened.

Having the same coordinates as the synchronized keyword, ReentrantLock follows the following idiom of implementation:

Lock / ReentrantLock lock = new ReentrantLock();  
...  
lock.lock();  
  
try {  
 ...  
} finally {  
 lock.unlock();  
}Copy

In the case of non-fair locks, the order in which threads are granted access is unspecified. If the lock should be fair (give precedence to the thread that has been waiting for the longest) then use the ReentrantLock​(boolean fair) constructor.

Summing up integers from 1 to 1,000,000 via ReentrantLock can be accomplished as follows:

public class CounterWithLock {  
  
 private static final Lock lock = new ReentrantLock();  
  
 private static int count;  
  
 public void counter() {  
 lock.lock();  
  
 try {  
 count++;  
 } finally {  
 lock.unlock();  
 }  
 }  
}Copy

And, let's use it via several threads:

CounterWithLock counterWithLock = new CounterWithLock();  
Runnable task = () -> {  
 counterWithLock.counter();  
};  
  
ExecutorService executor = Executors.newFixedThreadPool(8);  
for (int i = 0; i < 1 \_000\_000; i++) {  
 executor.execute(task);  
}Copy

Done!

As a bonus, the following code represents an idiom for resolving problems based on ReentrantLock.lockInterruptibly(). The code bundled to this book comes with an example of using lockInterruptibly():

Lock / ReentrantLock lock = new ReentrantLock();  
public void execute() throws InterruptedException {  
 lock.lockInterruptibly();  
  
 try {  
 // do something  
 } finally {  
 lock.unlock();  
 }  
}Copy

If the thread holding this lock is interrupted then InterruptedException is thrown. Using lock() instead of lockInterruptibly() will not be receptive to interruption.

In addition, the following code represents an idiom for using ReentrantLock.tryLock(long timeout, TimeUnit unit) throws InterruptedException. The code bundled to this book comes with an example as well:

Lock / ReentrantLock lock = new ReentrantLock();  
  
public boolean execute() throws InterruptedException {  
  
 if (!lock.tryLock(n, TimeUnit.SECONDS)) {  
 return false;  
 }  
  
 try {  
 // do something  
 } finally {  
 lock.unlock();  
 }  
  
 return true;  
}Copy

Note that tryLock() tries to acquire the lock for the specified time. If this time elapses, then the thread will not acquire the lock. It doesn't retry automatically. If the thread is interrupted during attempting to acquire the lock, then InterruptedException will be thrown.

Finally, the code bundled to this book comes with an example of using ReentrantLock.newCondition(). The idiom is in the next screenshot:

Texto

Descripción generada automáticamente

223. ReentrantReadWriteLock

Typically, a read-write tandem (for example, read-write a file) should be accomplished based on two statements:



Readers can read simultaneously as long as there are no writers (shared pessimistic lock).



A single writer can write at a time (exclusive/pessimistic locking).

The following diagram depicts readers on the left-hand side and writers on the right-hand side:

Diagrama

Descripción generada automáticamente con confianza media

Mainly, the following behavior is implemented by ReentrantReadWriteLock:



Provides pessimistic locking semantics for both locks (read and write lock).



If some readers hold the read lock and a writer wants the write lock, then no more readers are allowed to acquire the read lock until the writer released the write lock.



A writer can acquire the read lock, but a reader cannot acquire the write lock.

In case of non-fair locks, the order in which threads are granted access is unspecified. If the lock should be fair (give precedence to the thread that has been waiting for the longest), then use the ReentrantReadWriteLock​(boolean fair) constructor.

The idiom for using ReentrantReadWriteLock is shown as follows:

ReadWriteLock / ReentrantReadWriteLock lock   
 = new ReentrantReadWriteLock();  
...  
lock.readLock() / writeLock().lock();  
try {  
 ...  
} finally {  
 lock.readLock() / writeLock().unlock();  
}Copy

The following code represents a ReentrantReadWriteLock usage case that reads and writes an integer amount variable:

public class ReadWriteWithLock {  
  
 private static final Logger logger  
 = Logger.getLogger(ReadWriteWithLock.class.getName());  
 private static final Random rnd = new Random();  
  
 private static final ReentrantReadWriteLock lock  
 = new ReentrantReadWriteLock(true);  
  
 private static final Reader reader = new Reader();  
 private static final Writer writer = new Writer();  
  
 private static int amount;  
  
 private static class Reader implements Runnable {  
  
 @Override  
 public void run() {  
 if (lock.isWriteLocked()) {  
 logger.warning(() -> Thread.currentThread().getName()   
 + " reports that the lock is hold by a writer ...");  
 }  
  
 lock.readLock().lock();  
  
 try {  
 logger.info(() -> "Read amount: " + amount   
 + " by " + Thread.currentThread().getName());  
 } finally {  
 lock.readLock().unlock();  
 }  
 }  
 }  
  
 private static class Writer implements Runnable {  
  
 @Override  
 public void run() {  
 lock.writeLock().lock();  
 try {  
 Thread.sleep(rnd.nextInt(2000));  
 logger.info(() -> "Increase amount with 10 by "   
 + Thread.currentThread().getName());  
  
 amount += 10;  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 } finally {  
 lock.writeLock().unlock();  
 }  
 }  
 ...  
 }Copy

And, let's perform 10 reads and 10 writes with two readers and four writers:

ExecutorService readerService = Executors.newFixedThreadPool(2);  
ExecutorService writerService = Executors.newFixedThreadPool(4);  
  
for (int i = 0; i < 10; i++) {  
 readerService.execute(reader);  
 writerService.execute(writer);  
}Copy

A possible output will be as follows:

[09:09:25] [INFO] Read amount: 0 by pool-1-thread-1  
[09:09:25] [INFO] Read amount: 0 by pool-1-thread-2  
[09:09:26] [INFO] Increase amount with 10 by pool-2-thread-1  
[09:09:27] [INFO] Increase amount with 10 by pool-2-thread-2  
[09:09:28] [INFO] Increase amount with 10 by pool-2-thread-4  
[09:09:29] [INFO] Increase amount with 10 by pool-2-thread-3  
[09:09:29] [INFO] Read amount: 40 by pool-1-thread-2  
[09:09:29] [INFO] Read amount: 40 by pool-1-thread-1  
[09:09:31] [INFO] Increase amount with 10 by pool-2-thread-1  
...Copy

Before deciding to rely on ReentrantReadWriteLock, please consider that it may suffer from starvation (for example, when writers are given priority, readers might be starved). Moreover, we could not upgrade a read lock to a write lock (downgrading from writer to reader is possible), and there is no support for optimistic reads. If any of this matters for you then consider StampedLock, which we will look at in the next problem.

224. StampedLock

In a nutshell, StampedLock performs better than ReentrantReadWriteLock and supports optimistic reads. It is not like *reentrant;* therefore, it is prone to deadlocks. Mainly, a lock acquisition returns a stamp (a long value) that it is used in the finally block for unlocking. Each attempt to acquire a lock results in a new stamp, and, if no lock is available, then it may block until available. In other words, if the current thread is holding the lock, and attempts to acquire the lock again, it may cause a deadlock.

The StampedLock read/write orchestration process is achieved via several methods as follows:



readLock(): Non-exclusively acquires the lock, blocking if necessary, until available. For a non-blocking attempt of acquiring the read lock, we have to tryReadLock(). For timeout blocking, we have tryReadLock​(long time, TimeUnit unit). The returned stamp is used in unlockRead().



writeLock(): Exclusively acquires the lock, blocking if necessary until available. For a non-blocking attempt to acquire the write lock, we have tryWriteLock(). For timeout blocking, we have tryWriteLock​(long time, TimeUnit unit). The returned stamp is used in unlockWrite().



tryOptimisticRead(): This is the method that adds a big plus to StampedLock. This method returns a stamp that should be validated via the validate​() flag method. If the lock is not currently held in write mode, then the returned stamp is non-zero only.

The idioms for readLock() and writeLock() are pretty straightforward:

StampedLock lock = new StampedLock();  
...  
long stamp = lock.readLock() / writeLock();  
  
try {  
 ...  
} finally {  
 lock.unlockRead(stamp) / unlockWrite(stamp);  
}Copy

An attempt to give an idiom for tryOptimisticRead() can result in the following:

StampedLock lock = new StampedLock();  
  
int x; // a writer-thread can modify x  
...  
long stamp = lock.tryOptimisticRead();  
int thex = x;  
  
if (!lock.validate(stamp)) {  
 stamp = lock.readLock();  
  
 try {  
 thex = x;  
 } finally {  
 lock.unlockRead(stamp);  
 }  
}  
  
return thex;Copy

In this idiom, notice that the initial value (x) is assigned to the thex variable after getting the optimistic read lock. Then the validate() flag method is used to validate that the stamped lock has not been exclusively acquired since the emittance of the given stamp. If validate() returns false (equivalent with the fact that the write lock is acquired by a thread after the optimistic lock is acquired), then the read lock is acquired via the blocking readLock() and the value (x) is assigned again. Keep in mind that, if there is any write lock, the read lock may block. Acquiring the optimistic lock allows us to read the value(s) and then verify if there is any change in these value(s). Only if there is, will we have to go through the blocking read lock.

The following code represents a StampedLock usage case that reads and writes an integer amount variable. Basically, we reiterate the solution from the previous problem via optimistic reads:

public class ReadWriteWithStampedLock {  
  
 private static final Logger logger  
 = Logger.getLogger(ReadWriteWithStampedLock.class.getName());  
 private static final Random rnd = new Random();  
  
 private static final StampedLock lock = new StampedLock();  
  
 private static final OptimisticReader optimisticReader  
 = new OptimisticReader();  
 private static final Writer writer = new Writer();  
  
 private static int amount;  
  
 private static class OptimisticReader implements Runnable {  
  
 @Override  
 public void run() {  
 long stamp = lock.tryOptimisticRead();  
  
 // if the stamp for tryOptimisticRead() is not valid  
 // then the thread attempts to acquire a read lock  
 if (!lock.validate(stamp)) {  
 stamp = lock.readLock();  
 try {  
 logger.info(() -> "Read amount (read lock): " + amount   
 + " by " + Thread.currentThread().getName());  
 } finally {  
 lock.unlockRead(stamp);  
 }  
 } else {  
 logger.info(() -> "Read amount (optimistic read): " + amount   
 + " by " + Thread.currentThread().getName());  
 }  
 }  
 }  
  
 private static class Writer implements Runnable {  
  
 @Override  
 public void run() {  
  
 long stamp = lock.writeLock();  
  
 try {  
 Thread.sleep(rnd.nextInt(2000));  
 logger.info(() -> "Increase amount with 10 by "   
 + Thread.currentThread().getName());  
  
 amount += 10;  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 } finally {  
 lock.unlockWrite(stamp);  
 }  
 }  
 }  
 ...  
}Copy

And, let's perform 10 reads and 10 writes with two readers and four writers:

ExecutorService readerService = Executors.newFixedThreadPool(2);  
ExecutorService writerService = Executors.newFixedThreadPool(4);  
  
for (int i = 0; i < 10; i++) {  
 readerService.execute(optimisticReader);  
 writerService.execute(writer);  
}Copy

A possible output will be the following:

...  
[12:12:07] [INFO] Increase amount with 10 by pool-2-thread-4  
[12:12:07] [INFO] Read amount (read lock): 90 by pool-1-thread-2  
[12:12:07] [INFO] Read amount (optimistic read): 90 by pool-1-thread-2  
[12:12:07] [INFO] Increase amount with 10 by pool-2-thread-1  
...Copy

Starting with JDK 10, we can query the type of a stamp using isWriteLockStamp(), isReadLockStamp(), isLockStamp(), and isOptimisticReadStamp(). Based on the type, we can decide the proper unlock method, for example, as follows:

if (StampedLock.isReadLockStamp(stamp))  
 lock.unlockRead(stamp);  
}Copy

In the code bundled to this book, there is also an application for exemplifying the tryConvertToWriteLock​() method. In addition, you may be interested in developing applications that use tryConvertToReadLock​() and tryConvertToOptimisticRead().

225. Deadlock (dining philosophers)

What is deadlock? A famous joke on the internet explains it as follows:

*Interviewer:* Explain to us deadlock and we'll hire you!

*Me:* Hire me and I'll explain it to you ...

A simple deadlock can be explained as an A thread holding the L lock and trying to acquire the P lock, and, at the same time, there is a B thread holding the P lock and trying to acquire the L lock. This kind of deadlock is known as **circular wait**. Java doesn't have a deadlock detection and resolving mechanism (as databases have), and so a deadlock can be very embarrassing for the application. A deadlock can completely or partially block the application, can cause serious performance penalties, weird behaviors, and so on. Typically, deadlocks are hard to debug, and the only way to solve a deadlock consists of restarting the application and hoping for the best.

The dining philosophers is a famous problem used for illustrating a deadlock. This problem says that five philosophers are sitting around a table. Each of them alternates thinking and eating. In order to eat, a philosopher needs two forks in his hands—the fork from his left-hand side and the fork from his right-hand side. The difficulty is imposed by the fact that there are only five forks. After eating, the philosopher puts both forks back on the table, and they can then be picked up by another philosopher who repeats the same cycle. When a philosopher is not eating, he/she is thinking. The following diagram illustrates this scenario:

Forma, Círculo

Descripción generada automáticamente

The main task is to find a solution to this problem that allows the philosophers to think and eat in such a way so as to avoid being starved to death.

In the code, we can consider each philosopher as a Runnable instance. Being Runnable instances, we can execute them in separate threads. Each philosopher can pick up two forks placed to his left and right. If we represent a fork as a String, then we can use the following code:

public class Philosopher implements Runnable {  
  
 private final String leftFork;  
 private final String rightFork;  
  
 public Philosopher(String leftFork, String rightFork) {  
 this.leftFork = leftFork;  
 this.rightFork = rightFork;  
 }  
  
 @Override  
 public void run() {  
 // implemented below  
 }  
}Copy

So, a philosopher can pick up leftFork and rightFork. But since the philosophers share these forks, a philosopher must acquire exclusive locks on these two forks. Having an exclusive lock on leftFork and an exclusive lock on rightFork is equivalent to having two forks in his hands. Having exclusive locks on leftFork and rightFork is equivalent to the philosopher eating. Releasing both exclusive locks is equivalent to the philosopher not eating and thinking.

Locking can be achieved via the synchronized keyword as in the following run() method:

@Override  
public void run() {  
  
 while (true) {  
 logger.info(() -> Thread.currentThread().getName()   
 + ": thinking");  
 doIt();  
  
 synchronized(leftFork) {  
 logger.info(() -> Thread.currentThread().getName()   
 + ": took the left fork (" + leftFork + ")");  
 doIt();  
  
 synchronized(rightFork) {  
 logger.info(() -> Thread.currentThread().getName()   
 + ": took the right fork (" + rightFork + ") and eating");  
 doIt();  
  
 logger.info(() -> Thread.currentThread().getName()   
 + ": put the right fork ( " + rightFork   
 + ") on the table");  
 doIt();  
 }  
  
 logger.info(() -> Thread.currentThread().getName()   
 + ": put the left fork (" + leftFork   
 + ") on the table and thinking");  
 doIt();  
 }  
 }  
}Copy

A philosopher starts by thinking. After a while he is hungry, so he tries to pick up the left and right forks. If successful he will eat for a while. Afterwards, he put the forks on the table and continues to think until he is hungry again. Meanwhile, another philosopher will eat.

The doIt() method simulates the involved actions (thinking, eating, picking, and putting back the forks) via a random sleep. This can be seen in the code as follows:

private static void doIt() {  
 try {  
 Thread.sleep(rnd.nextInt(2000));  
 } catch (InterruptedException ex) {  
 Thread.currentThread().interrupt();  
 logger.severe(() -> "Exception: " + ex);  
 }  
}Copy

Finally, we need forks and the philosophers, see the following code:

String[] forks = {  
 "Fork-1", "Fork-2", "Fork-3", "Fork-4", "Fork-5"  
};  
  
Philosopher[] philosophers = {  
 new Philosopher(forks[0], forks[1]),  
 new Philosopher(forks[1], forks[2]),  
 new Philosopher(forks[2], forks[3]),  
 new Philosopher(forks[3], forks[4]),  
 new Philosopher(forks[4], forks[0])  
};Copy

Each philosopher will run in a thread, as follows:

Thread threadPhilosopher1   
 = new Thread(philosophers[0], "Philosopher-1");  
...  
Thread threadPhilosopher5   
 = new Thread(philosophers[4], "Philosopher-5");  
  
threadPhilosopher1.start();  
...  
threadPhilosopher5.start();Copy

This implementation seems to be OK and may even work fine for a while. However, sooner or later this implementation blocks with output as follows:

[17:29:21] [INFO] Philosopher-5: took the left fork (Fork-5)  
...  
// nothing happensCopy

This is a deadlock! Each philosopher has his left fork in hand (exclusive lock on it) and waits for the right fork to be on the table (the lock is to be released). Obviously, this expectation cannot be satisfied, since there are only five forks and each philosopher has one in his hands.

In order to avoid this deadlock, there is a pretty simple solution. We just force one of the philosophers to pick up the right fork first. After successfully picking the right fork, he can try to pick the left one. In the code, this is a quick modification to the following line:

// the original line  
new Philosopher(forks[4], forks[0])  
  
// the modified line that eliminates the deadlock  
new Philosopher(forks[0], forks[4])Copy

This time we can run the application without deadlocks.

Summary

Well, that's all! This chapter covered problems about the fork/join framework, CompletableFuture, ReentrantLock, ReentrantReadWriteLock, StampedLock, atomic variables, tasks cancellation, interruptible methods, thread-local, and deadlocks.

Download the applications from this chapter to see the results and to see additional details.

Optional

This chapter includes 24 problems meant to draw your attention to several rules for working with Optional. The problems and solutions presented in this section are based on the Java language architect Brian Goetz's definition:

*"Optional is intended to provide a limited mechanism for library method return types where there needed to be a clear way to represent no result, and using null for such was overwhelmingly likely to cause errors*."

But where there are rules, there are exceptions as well. Therefore, do not conclude that the rules (or practices) presented here should be followed (or avoided) at all costs. As always, it depends on the problem, and you have to evaluate the situation, weighing up the pros and cons.

You may also like to check the CDI plugin (<https://github.com/Pscheidl/FortEE>) for Java EE (Jakarta EE) developed by Pavel Pscheidl. This is a Jakarta EE/Java EE fault-tolerance guard leveraging the Optional pattern. Its power lies in its simplicity.

Problems

Use the following problems to test your Optional programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **Initializing Optional**: Write a program that exemplifies the right and wrong approaches for initializing Optional.
2. **Optional.get() and missing value**: Write a program that exemplifies the right and wrong usage of Optional.get().
3. **Returning an already-constructed default value**: Write a program that, when no value is present, sets (or returns) an already-constructed default value via the Optional.orElse() method.
4. **Returning a non-existent default value**: Write a program that, when no value is present, sets (or returns) a non-existent default value via the Optional.orElseGet() method.
5. **Throwing NoSuchElementException**: Write a program that, when no value is present, throws an exception of the NoSuchElementException type or another exception.
6. **The Optional and null references**: Write a program that exemplifies the correct usage of Optional.orElse(null).
7. **Consuming a present Optional class**: Write a program that consumes a present Optional class via ifPresent() and via ifPresentElse().
8. **Returning a present Optional class or another one**: Let's assume that we have Optional. Write a program that relies on Optional.or() for returning this Optional (if its value is present) or another Optional class (if its value is not present).
9. **Chaining lambdas via** orElseFoo(): Write a program that exemplifies the usage of orElse() and orElseFoo() for avoiding disrupting lambda chains.
10. **Do not use Optional just for getting a value**: Exemplify the bad practice of chaining the Optional methods with the single purpose of getting some values.
11. **Do not use Optional for fields**: Exemplify the bad practice of declaring fields of the Optional type.
12. **Do not use Optional in constructor args**: Exemplify the bad practice of using Optional in constructors arguments.
13. **Do not use Optional in setters args**: Exemplify the bad practice of using Optional in setter arguments.
14. **Do not use Optional in methods args**: Exemplify the bad practice of using Optional in method arguments.
15. **Do not use Optional to return empty or null collections or arrays**: Exemplify the bad practice of using Optional for returning the empty/null collections or arrays.
16. **Avoiding Optional in collections**: Using Optional in collections can be a design smell. Exemplify a typical use case and possible alternatives for avoiding Optional in collections.
17. **Confusing of() with ofNullable()**: Exemplify the potential consequences of confusing Optional.of() with ofNullable().
18. **Optional<T> versus OptionalInt**: Exemplify the usage of non-generic OptionalInt instead of Optional<T>.
19. **Asserting equality of Optional classes**: Exemplify asserting the equality of Optional classes.
20. **Transforming values via map() and flatMap()**: Write several snippets of code for exemplifying the usage of Optional.map() and flatMap().
21. **Filter values via Optional.filter()**: Exemplify the usage of Optional.filter() for rejecting wrapped values based on a predefined rule.
22. **Chaining the Optional and Stream APIs**: Exemplify the usage of Optional.stream() for chaining the Optional API with the Stream API.
23. **Optional and identity-sensitive operations**: Write a snippet of code that sustains the fact that *identity-sensitive* operations should be avoided in the case of Optional.
24. **Return boolean if Optional is empty**: Write two snippets of code for exemplifying two solutions for returning boolean if the given Optional class is empty.

Solutions

The following sections describe solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations shown here include only the most interesting and important details needed to solve the problems. Download the example solutions to see additional details and to experiment with the programs at <https://github.com/PacktPublishing/Java-Coding-Problems>.

226. Initializing Optional

Initializing Optional should be done via Optional.empty() instead of null:

// Avoid  
Optional<Book> book = null;  
  
// Prefer  
Optional<Book> book = Optional.empty();Copy

Since Optional acts as a container (box), it is meaningless to initialize it with null.

227. Optional.get() and missing value

So, if we have decided to call Optional.get() to fetch the value wrapped in Optional, then we shouldn't do it as follows:

Optional<Book> book = ...; // this is prone to be empty  
  
// Avoid  
// if "book" is empty then the following code will  
// throw a java.util.NoSuchElementException  
Book theBook = book.get();Copy

In other words, before fetching the value via Optional.get(), we need to prove that the value is present. A solution consists of calling isPresent() before calling get(). This way, we add a check that allows us to handle the missing value case:

Optional<Book> book = ...; // this is prone to be empty  
  
// Prefer  
if (book.isPresent()) {  
 Book theBook = book.get();  
 ... // do something with "theBook"  
} else {  
 ... // do something that does not call book.get()  
}Copy

Nevertheless, keep in mind that the isPresent()-get() team has a bad reputation, and so use it with caution. Consider checking the next problems, which provide alternatives to this team. Moreover, at some point, Optional.get() is likely to be deprecated.

228. Returning an already-constructed default value

Let's assume that we have a method that returns a result based on Optional. If Optional is empty then the method returns a default value. If we consider the previous problem, then a possible solution can be written as follows:

public static final String BOOK\_STATUS = "UNKNOWN";  
...  
// Avoid  
public String findStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 if (status.isPresent()) {  
 return status.get();  
 } else {  
 return BOOK\_STATUS;  
 }  
}Copy

Well, this is not a bad solution, but is not very elegant. A more concise and elegant solution will rely on the Optional.orElse() method. This method is useful for replacing the isPresent()-get() pair when we want to set or return a default value in case of an empty Optional class. The preceding snippet of code can be rewritten as follows:

public static final String BOOK\_STATUS = "UNKNOWN";  
...  
// Prefer  
public String findStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 return status.orElse(BOOK\_STATUS);  
}Copy

But keep in mind that orElse() is evaluated even when the Optional class involved is not empty. In other words, orElse() is evaluated even if its value is not used. Having said that, it is advisable to rely on orElse() only when its argument is an already-constructed value. That way, we mitigate a potential performance penalty. The next problem addresses the case when orElse() is not the correct choice.

229. Returning a non-existent default value

Let's assume that we have a method that returns a result based on an Optional class. If this Optional class is empty then the method returns a computed value. The computeStatus() method computes this value:

private String computeStatus() {  
 // some code used to compute status  
}Copy

Now, a clumsy solution will rely on the isPresent()-get() pair, as follows:

// Avoid  
public String findStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 if (status.isPresent()) {  
 return status.get();  
 } else {  
 return computeStatus();  
 }  
}Copy

Even if this solution is clumsy, it is still better than relying on the orElse() method, as follows:

// Avoid  
public String findStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 // computeStatus() is called even if "status" is not empty  
 return status.orElse(computeStatus());  
}Copy

In this case, the preferred solution relies on the Optional.orElseGet() method. The argument of this method is Supplier;, and so it is executed only when the Optional value is not present. This is much better than orElse() since it saves us from executing extra code that shouldn't be executed when the Optional value is present. So, the preferred solution is as follows:

// Prefer  
public String findStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 // computeStatus() is called only if "status" is empty  
 return status.orElseGet(this::computeStatus);  
}Copy

230. Throwing NoSuchElementException

Sometimes, if Optional is empty, we want to throw an exception (for example, NoSuchElementException). The clumsy solution to this problem is listed as follows:

// Avoid  
public String findStatus() {  
  
 Optional<String> status = ...; // this is prone to be empty  
  
 if (status.isPresent()) {  
 return status.get();  
 } else {  
 throw new NoSuchElementException("Status cannot be found");  
 }  
}Copy

But a much more elegant solution will rely on the Optional.orElseThrow() method. The signature of this method, orElseThrow(Supplier<? extends X> exceptionSupplier), allows us to give the exception as follows (if the value is present then orElseThrow() will return it):

// Prefer  
public String findStatus() {  
  
 Optional<String> status = ...; // this is prone to be empty  
  
 return status.orElseThrow(  
 () -> new NoSuchElementException("Status cannot be found"));  
}Copy

Or, another exception is, for example, IllegalStateException:

// Prefer  
public String findStatus() {  
  
 Optional<String> status = ...; // this is prone to be empty  
  
 return status.orElseThrow(  
 () -> new IllegalStateException("Status cannot be found"));  
}Copy

Starting with JDK 10, Optional was enriched with an orElseThrow() flavor without arguments. This method implicitly throws NoSuchElementException:

// Prefer (JDK 10+)  
public String findStatus() {  
  
 Optional<String> status = ...; // this is prone to be empty  
  
 return status.orElseThrow();  
}Copy

Nevertheless, be aware that throwing an unchecked exception without a meaningful message in production is not good practice.

231. Optional and null references

It is possible to take advantage of orElse(null) by using a method that accepts the null references in certain situations.

A candidate for this scenario is Method.invoke() from the Java Reflection API (see [Chapter 7](https://subscription.packtpub.com/book/programming/9781789801415/12/ch12lvl1sec48/249d02dd-04fc-4b26-ba01-a9a47543c7e7.xhtml), *Java Reflection Classes, Interfaces, Constructors, Methods, and Fields*).

The first argument of Method.invoke() represents the object instance on which this particular method is to be invoked. If the method is static, the first argument should be null, and so there is no need to have an instance of the object.

Let's assume that we have a class named Book and the helper method listed as follows.

This method returns an empty Optional class (if the given method is static) or an Optional class containing an instance of Book (if the given method is non-static):

private static Optional<Book> fetchBookInstance(Method method) {  
  
 if (Modifier.isStatic(method.getModifiers())) {  
 return Optional.empty();  
 }  
  
 return Optional.of(new Book());  
}Copy

Calling this method is pretty simple:

Method method = Book.class.getDeclaredMethod(...);  
  
Optional<Book> bookInstance = fetchBookInstance(method);Copy

Furthermore, if Optional is empty (meaning that the method is static), we need to pass null to Method.invoke(); otherwise, we pass the Book instance. A clumsy solution may rely on the isPresent()-get() pair, as follows:

// Avoid  
if (bookInstance.isPresent()) {  
 method.invoke(bookInstance.get());  
} else {  
 method.invoke(null);  
}Copy

But this is a perfect fit for Optional.orElse(null). The following code reduces the solution to a single line of code:

// Prefer  
method.invoke(bookInstance.orElse(null));Copy

As a rule of thumb, we should use orElse(null) only when we have Optional and we need a null reference. Otherwise, avoid orElse(null).

232. Consuming a present Optional class

Sometimes, all we want is to consume a present Optional class. If Optional is not present then nothing needs to be done. An unskillful solution will rely on the isPresent()-get() pair, as follows:

// Avoid  
public void displayStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 if (status.isPresent()) {  
 System.out.println(status.get());  
 }  
}Copy

A better solution relies on ifPresent(), which takes Consumer as an argument. This is an alternative to the isPresent()-get() pair when we just need to consume the present value. The code can be rewritten as follows:

// Prefer  
public void displayStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 status.ifPresent(System.out::println);  
}Copy

But in other cases, if Optional is not present then we want to execute an empty-based action. The solution based on the isPresent()-get() pair is as follows:

// Avoid  
public void displayStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 if (status.isPresent()) {  
 System.out.println(status.get());  
 } else {  
 System.out.println("Status not found ...");  
 }  
}Copy

Again, this is not the best choice. Alternatively, we can count on ifPresentOrElse(). This method has been available since JDK 9 and is similar to the ifPresent() method; the only difference is that it covers the else branch as well:

// Prefer  
public void displayStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 status.ifPresentOrElse(System.out::println,  
 () -> System.out.println("Status not found ..."));  
}Copy

233. Returning a present Optional class or another one

Let's consider a method that returns an Optional class. Mainly, this method computes an Optional class and, if it isn't empty, then it simply returns this Optional class. Otherwise, if the computed Optional class is empty then we execute some other action that also returns Optional class.

The isPresent()-get() pair can do it as follows (this should be avoided):

private final static String BOOK\_STATUS = "UNKNOWN";  
...  
// Avoid  
public Optional<String> findStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 if (status.isPresent()) {  
 return status;  
 } else {  
 return Optional.of(BOOK\_STATUS);  
 }  
}Copy

Alternatively, we should avoid such constructions as follow:

return Optional.of(status.orElse(BOOK\_STATUS));  
return Optional.of(status.orElseGet(() -> (BOOK\_STATUS)));Copy

The best solution has been available since with JDK 9, and it consists of the Optional.or() method. This method is capable of returning Optional describing the value. Otherwise, it returns Optional produced by the given Supplier function (the supplying function that produces Optional to be returned):

private final static String BOOK\_STATUS = "UNKNOWN";  
...  
// Prefer  
public Optional<String> findStatus() {  
 Optional<String> status = ...; // this is prone to be empty  
  
 return status.or(() -> Optional.of(BOOK\_STATUS));  
}Copy

234. Chaining lambdas via orElseFoo()

Some operations specific to lambda expressions return Optional (for example, findFirst(), findAny(), reduce(), and so on). Trying to address these Optional classes via the isPresent()-get() pair is a cumbersome solution because we have to break the chain of lambdas, add some conditional code via the if-else blocks, and consider resuming the chain.

The following snippet of code shows this practice:

private static final String NOT\_FOUND = "NOT FOUND";  
  
List<Book> books...;  
...  
// Avoid  
public String findFirstCheaperBook(int price) {  
  
 Optional<Book> book = books.stream()  
 .filter(b -> b.getPrice()<price)  
 .findFirst();  
  
 if (book.isPresent()) {  
 return book.get().getName();  
 } else {  
 return NOT\_FOUND;  
 }  
}Copy

One step further and we may have something like the following:

// Avoid  
public String findFirstCheaperBook(int price) {  
  
 Optional<Book> book = books.stream()  
 .filter(b -> b.getPrice()<price)  
 .findFirst();  
  
 return book.map(Book::getName)  
 .orElse(NOT\_FOUND);  
}Copy

Using orElse() instead of the isPresent()-get() pair is better. But it will be even better if we use orElse() (and orElseFoo()) directly in the chain of lambdas and avoid disrupted code:

private static final String NOT\_FOUND = "NOT FOUND";  
...  
// Prefer  
public String findFirstCheaperBook(int price) {  
  
 return books.stream()  
 .filter(b -> b.getPrice()<price)  
 .findFirst()  
 .map(Book::getName)  
 .orElse(NOT\_FOUND);  
}Copy

Let's have one more problem.

This time, we have an author of several books, and we want to check whether a certain book was written by this author. If our author didn't write the given book, then we want to throw NoSuchElementException.

A really bad solution to this will be as follows:

// Avoid  
public void validateAuthorOfBook(Book book) {  
 if (!author.isPresent() ||  
 !author.get().getBooks().contains(book)) {  
 throw new NoSuchElementException();  
 }  
}Copy

On the other hand, using orElseThrow() can solve the problem very elegantly:

// Prefer  
public void validateAuthorOfBook(Book book) {  
 author.filter(a -> a.getBooks().contains(book))  
 .orElseThrow();  
}Copy

235. Do not use Optional just for getting a value

This problem gives the start of a suite of problems from the *do not use* category. The *do not use* category tries to prevent the *overuse* of Optional and gives several rules that can save us a lot of trouble. Nevertheless, rules have exceptions. Therefore, do not conclude that the presented rules should be avoided at all costs. As always, it depends on the problem.

In the case of Optional, a common scenario involves chaining its methods for the single purpose of getting some value.

Avoid this practice and rely on simple and straightforward code. In other words, avoid doing something like the following snippet of code:

public static final String BOOK\_STATUS = "UNKNOWN";  
...  
// Avoid  
public String findStatus() {  
 // fetch a status prone to be null  
 String status = ...;  
  
 return Optional.ofNullable(status).orElse(BOOK\_STATUS);  
}Copy

And use a simple if-else block or the ternary operator (for simple cases):

// Prefer  
public String findStatus() {  
 // fetch a status prone to be null  
 String status = null;  
  
 return status == null ? BOOK\_STATUS : status;  
}Copy

236. Do not use Optional for fields

The *do not use* category continues with the following statement—Optional was not intended to be used for fields and it doesn't implement Serializable.

The Optional class is definitively not intended to be used as a field of a JavaBean. So, do not do this:

// Avoid  
public class Book {  
  
 [access\_modifier][static][final]  
 Optional<String> title;  
 [access\_modifier][static][final]  
 Optional<String> subtitle = Optional.empty();  
 ...  
}Copy

But do this:

// Prefer  
public class Book {  
  
 [access\_modifier][static][final] String title;  
 [access\_modifier][static][final] String subtitle = "";  
 ...  
}Copy

237. Do not use Optional in constructor args

The *do not use* category continues with another scenario that is against the intention of using Optional. Keep in mind that Optional represents a container for objects; therefore, Optional adds another level of abstraction. In other words, improper use of Optional simply adds extra *boilerplate* code.

Check the following use case of Optional that shows this (this code violates the previous *Do not use Optional for fields* section):

// Avoid  
public class Book {  
  
 // cannot be null  
 private final String title;   
  
 // optional field, cannot be null  
 private final Optional<String> isbn;  
  
 public Book(String title, Optional<String> isbn) {  
 this.title = Objects.requireNonNull(title,  
 () -> "Title cannot be null");  
  
 if (isbn == null) {  
 this.isbn = Optional.empty();  
 } else {  
 this.isbn = isbn;  
 }  
  
 // or  
 this.isbn = Objects.requireNonNullElse(isbn, Optional.empty());  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public Optional<String> getIsbn() {  
 return isbn;  
 }  
}Copy

We can fix this code by removing Optional from the fields and from the constructor arguments, as follows:

// Prefer  
public class Book {  
  
 private final String title; // cannot be null  
 private final String isbn; // can be null  
  
 public Book(String title, String isbn) {  
 this.title = Objects.requireNonNull(title,  
 () -> "Title cannot be null");  
 this.isbn = isbn;  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public Optional<String> getIsbn() {  
 return Optional.ofNullable(isbn);  
 }  
}Copy

The getter of isbn returns Optional. But do not consider this example as a rule for transforming all of your getters in this way. Some getters return collections or arrays, and, in that case, they prefer returning empty collections/arrays instead of Optional. Use this technique and keep in mind the statement of Brian Goetz (Java's language architect):

*"I think routinely using it as a return value for getters would definitely be over-use."*

*- Brian Goetz*

238. Do not use Optional in setter args

The *do not use* category continues with a very tempting scenario that consists of using Optional in setter arguments. The following code should be avoided since it adds extra boilerplate code and violates the *Do not use Optional for fields* section (check the setIsbn() method):

// Avoid  
public class Book {  
  
 private Optional<String> isbn;  
  
 public Optional<String> getIsbn() {  
 return isbn;  
 }  
  
 public void setIsbn(Optional<String> isbn) {  
 if (isbn == null) {  
 this.isbn = Optional.empty();  
 } else {  
 this.isbn = isbn;  
 }  
  
 // or  
 this.isbn = Objects.requireNonNullElse(isbn, Optional.empty());  
 }  
}Copy

We can fix this code by removing Optional from the fields and from the setters' arguments as follows:

// Prefer  
public class Book {  
  
 private String isbn;  
  
 public Optional<String> getIsbn() {  
 return Optional.ofNullable(isbn);  
 }  
  
 public void setIsbn(String isbn) {  
 this.isbn = isbn;  
 }  
}Copy

Commonly, this bad practice is used in JPA entities for persistent properties (to map an entity attribute as Optional). However, using Optional in domain model entities is possible.

239. Do not use Optional in method args

The *do not use* category continues with another common mistake of using Optional. This time let's address the usage of Optional in method arguments.

Using Optional in method arguments is just another use case that may lead to code that is unnecessarily complicated. Mainly, it is advisable to take responsibility for null-checking arguments instead of trusting that the callers will create Optional classes, especially empty Optional classes. This bad practice clutters the code and is still prone to NullPointerException. The caller can still pass null. So you have just turned back to checking null arguments.

Keep in mind that Optional is just another object (a container) and is not cheap. Optional consumes four times the memory of a bare reference!

As a conclusion, think twice before doing something like the following:

// Avoid  
public void renderBook(Format format,  
 Optional<Renderer> renderer, Optional<String> size) {  
  
 Objects.requireNonNull(format, "Format cannot be null");  
  
 Renderer bookRenderer = renderer.orElseThrow(  
 () -> new IllegalArgumentException("Renderer cannot be empty")  
 );  
  
 String bookSize = size.orElseGet(() -> "125 x 200");  
 ...  
}Copy

Check the following call of this method that creates the required Optional class. But, obviously, passing null is possible as well and will result in NullPointerException, but this means that you intentionally defeat the purpose of Optional—don't think of polluting the preceding code with null checks for the Optional parameters; that would be a really bad idea:

Book book = new Book();  
  
// Avoid  
book.renderBook(new Format(),  
 Optional.of(new CoolRenderer()), Optional.empty());  
  
// Avoid  
// lead to NPE  
book.renderBook(new Format(),  
 Optional.of(new CoolRenderer()), null);Copy

We can fix this code by removing Optional classes as follows:

// Prefer  
public void renderBook(Format format,   
 Renderer renderer, String size) {  
  
 Objects.requireNonNull(format, "Format cannot be null");  
 Objects.requireNonNull(renderer, "Renderer cannot be null");  
  
 String bookSize = Objects.requireNonNullElseGet(  
 size, () -> "125 x 200");  
 ...  
}Copy

This time, the call of this method doesn't force the creation of Optional:

Book book = new Book();  
  
// Prefer  
book.renderBook(new Format(), new CoolRenderer(), null);Copy

When a method can accept optional parameters, rely on old-school method overloading, not on Optional.

240. Do not use Optional to return empty or null collections or arrays

Furthermore, in the *do not use* category, let's tackle the usage of Optional as the return type that wraps an empty or null collection or array.

Returning Optional that wraps an empty or null collection/array may be comprised of a clean and lightweight code. Check out the following code that shows this:

// Avoid  
public Optional<List<Book>> fetchBooksByYear(int year) {  
 // fetching the books may return null  
 List<Book> books = ...;  
  
 return Optional.ofNullable(books);  
}  
  
Optional<List<Book>> books = author.fetchBooksByYear(2021);  
  
// Avoid  
public Optional<Book[]> fetchBooksByYear(int year) {  
 // fetching the books may return null  
 Book[] books = ...;  
  
 return Optional.ofNullable(books);  
}  
  
Optional<Book[]> books = author.fetchBooksByYear(2021);Copy

We can clean this code by removing the unnecessary Optional and then rely on empty collections (for example, Collections.emptyList(), emptyMap(), and emptySet()) and arrays (for example, new String[0]). This is the preferable solution:

// Prefer  
public List<Book> fetchBooksByYear(int year) {  
 // fetching the books may return null  
 List<Book> books = ...;  
  
 return books == null ? Collections.emptyList() : books;  
}  
  
List<Book> books = author.fetchBooksByYear(2021);  
  
// Prefer  
public Book[] fetchBooksByYear(int year) {  
 // fetching the books may return null  
 Book[] books = ...;  
  
 return books == null ? new Book[0] : books;  
}  
  
Book[] books = author.fetchBooksByYear(2021);Copy

If you need to distinguish between a missing and an empty collection/array then throw an exception for the missing.

241. Avoiding Optional in collections

Relying on Optional in collections can be a design smell. Take another 30 minutes to re-evaluate the problem and discover better solutions.

The preceding statement is valid especially in the case of Map when the reason behind this decision sounds like this—so, Map returns null if there is no mapping for a key or if null is mapped to the key, so I cannot tell whether the key is not present or is a missing value. I will wrap the values via Optional.ofNullable() and done!

But what will we decide further if Map of Optional<Foo> is populated with null values, absent Optional values, or even Optional objects that contain something else, but not *Foo*? Haven't we just nested the initial problem into one more layer? How about the performance penalty? Optional is not cost free; it is just another object that consumes memory and needs to be collected.

So, let's consider a solution that should be avoided:

private static final String NOT\_FOUND = "NOT FOUND";  
...  
// Avoid  
Map<String, Optional<String>> isbns = new HashMap<>();  
isbns.put("Book1", Optional.ofNullable(null));  
isbns.put("Book2", Optional.ofNullable("123-456-789"));  
...  
Optional<String> isbn = isbns.get("Book1");  
  
if (isbn == null) {  
 System.out.println("This key cannot be found");  
} else {  
 String unwrappedIsbn = isbn.orElse(NOT\_FOUND);  
 System.out.println("Key found, Value: " + unwrappedIsbn);  
}Copy

A better and elegant solution can rely on JDK 8, getOrDefault() as follows:

private static String get(Map<String, String> map, String key) {  
 return map.getOrDefault(key, NOT\_FOUND);  
}  
  
Map<String, String> isbns = new HashMap<>();  
isbns.put("Book1", null);  
isbns.put("Book2", "123-456-789");  
...  
String isbn1 = get(isbns, "Book1"); // null  
String isbn2 = get(isbns, "Book2"); // 123-456-789  
String isbn3 = get(isbns, "Book3"); // NOT FOUNDCopy

Other solutions can rely on the following:

* The containsKey() method
* Trivial implementation by extending HashMap
* The JDK 8 computeIfAbsent() method
* Apache Commons DefaultedMap

We can conclude that there are always better solutions than using Optional in collections.

But the discussed use case from earlier is not the worst-case scenario. Here are two more that must be avoided:

Map<Optional<String>, String> items = new HashMap<>();  
Map<Optional<String>, Optional<String>> items = new HashMap<>();Copy

242. Confusing of() with ofNullable()

Confusing or mistakenly using Optional.of() instead of Optional.ofNullable(), or vice versa, can lead to weird behaviors and even NullPointerException.

Optional.of(null) will throw NullPointerException, while Optional.ofNullable(null) will result in Optional.empty.

Check the following failed attempt to write a snippet of code to avoid NullPointerException:

// Avoid  
public Optional<String> isbn(String bookId) {  
 // the fetched "isbn" can be null for the given "bookId"  
 String isbn = ...;  
  
 return Optional.of(isbn); // this throws NPE if "isbn" is null :(  
}Copy

But, most probably, we actually wanted to use ofNullable(), as follows:

// Prefer  
public Optional<String> isbn(String bookId) {  
 // the fetched "isbn" can be null for the given "bookId"  
 String isbn = ...;  
  
 return Optional.ofNullable(isbn);  
}Copy

Using ofNullable() instead of of() is not a disaster, but it may cause some confusion and bring no value. Check the following code:

// Avoid  
// ofNullable() doesn't add any value  
return Optional.ofNullable("123-456-789");  
  
// Prefer  
return Optional.of("123-456-789"); // no risk to NPECopy

Here is another problem. Let's assume that we want to convert an empty String object into an empty Optional. We may think that the proper solution will rely on of(), as follows:

// Avoid  
Optional<String> result = Optional.of(str)  
 .filter(not(String::isEmpty));Copy

But remember that String can be null. This solution will work fine for empty or non-empty strings, but not for the null strings. Therefore, ofNullable() gives us the proper solution, as follows:

// Prefer  
Optional<String> result = Optional.ofNullable(str)  
 .filter(not(String::isEmpty));Copy

243. Optional<T> versus OptionalInt

If there is no specific reason for using *boxed primitives*, then it is advisable to avoid Optional<T> and rely on non-generic OptionalInt, OptionalLong, or OptionalDouble type.

Boxing and unboxing are expensive operations that are prone to induce performance penalties. In order to eliminate this risk, we can rely on OptionalInt, OptionalLong, and OptionalDouble. These are wrappers for the int, long, and double primitive types.

So, avoid the following (and similar) solutions:

// Avoid  
Optional<Integer> priceInt = Optional.of(50);  
Optional<Long> priceLong = Optional.of(50L);  
Optional<Double> priceDouble = Optional.of(49.99d);Copy

And prefer the following solutions:

// Prefer  
// unwrap via getAsInt()  
OptionalInt priceInt = OptionalInt.of(50);  
  
// unwrap via getAsLong()  
OptionalLong priceLong = OptionalLong.of(50L);  
  
// unwrap via getAsDouble()  
OptionalDouble priceDouble = OptionalDouble.of(49.99d);Copy

244. Asserting equality of Optionals

Having two Optional objects in assertEquals() doesn't require unwrapped values. This is applicable because Optional.equals() compares the wrapped values, not the Optional objects. This is the source code of Optional.equals():

@Override  
public boolean equals(Object obj) {  
  
 if (this == obj) {  
 return true;  
 }  
  
 if (!(obj instanceof Optional)) {  
 return false;  
 }  
  
 Optional<?> other = (Optional<?>) obj;  
  
 return Objects.equals(value, other.value);  
}Copy

Let's assume that we have two Optional objects:

Optional<String> actual = ...;  
Optional<String> expected = ...;  
  
// or  
Optional actual = ...;  
Optional expected = ...; Copy

It is advisable to avoid a test written as follows:

// Avoid  
@Test  
public void givenOptionalsWhenTestEqualityThenTrue()   
 throws Exception {  
  
 assertEquals(expected.get(), actual.get());  
}Copy

If expected and/or actual is empty, then the get() method will cause an exception of the NoSuchElementException type.

It is better to use the following test:

// Prefer  
@Test  
public void givenOptionalsWhenTestEqualityThenTrue()   
 throws Exception {  
  
 assertEquals(expected, actual);  
}Copy

245. Transforming values via Map() and flatMap()

The Optional.map() and flatMap() methods are convenient for transforming an Optional value.

The map() method applies the function argument to the value, then returns the result wrapped in an Optional object. The flatMap() method applies the function argument to the value and then returns the result directly.

Let's assume that we have Optional<String>, and we want to transform this String from lowercase into uppercase. An uninspired solution can be written as follows:

Optional<String> lowername = ...; // may be empty as well  
  
// Avoid  
Optional<String> uppername;  
  
if (lowername.isPresent()) {  
 uppername = Optional.of(lowername.get().toUpperCase());  
} else {  
 uppername = Optional.empty();  
}Copy

A more inspired solution (in a single line of code) will rely on Optional.map(), as follows:

// Prefer  
Optional<String> uppername = lowername.map(String::toUpperCase);Copy

The map() method can be useful to avoid breaking a chain of lambdas as well. Let's consider List<Book>, and we want to find the first book that's $50 cheaper and, if such a book exists, change its title to uppercase. Again, an uninspired solution will be as follows:

private static final String NOT\_FOUND = "NOT FOUND";  
List<Book> books = Arrays.asList();  
...  
// Avoid  
Optional<Book> book = books.stream()  
 .filter(b -> b.getPrice()<50)  
 .findFirst();  
  
String title;  
if (book.isPresent()) {  
 title = book.get().getTitle().toUpperCase();  
} else {  
 title = NOT\_FOUND;  
}Copy

Relying on map(), we can do it via the following chain of lambdas:

// Prefer  
String title = books.stream()  
 .filter(b -> b.getPrice()<50)  
 .findFirst()  
 .map(Book::getTitle)  
 .map(String::toUpperCase)  
 .orElse(NOT\_FOUND);Copy

In the preceding example, the getTitle() method is a classical getter that returns the title of the book as String. But let's modify this getter to return Optional:

public Optional<String> getTitle() {  
 return ...;  
}Copy

This time, we cannot use map() because map(Book::getTitle) will return Optional<Optional<String>> instead of Optional<String>. But if we rely on flatMap(), then the return of it will not be wrapped in an additional Optional object:

// Prefer  
String title = books.stream()  
 .filter(b -> b.getPrice()<50)  
 .findFirst()  
 .flatMap(Book::getTitle)  
 .map(String::toUpperCase)  
 .orElse(NOT\_FOUND);Copy

So, Optional.map() wraps the result of transformation in an Optional object. If this result is Optional itself, then we obtain Optional<Optional<...>>. On the other hand, flatMap() does not wrap the result within an additional Optional object.

246. Filter values via Optional.filter()

Using Optional.filter() to accept or reject a wrapped value is a very convenient approach since it can be accomplished without explicitly unwrapping the value. We just pass a predicate (the condition) as an argument and get an Optional object (the initial Optional object if the condition is met or an empty Optional object if the condition is not met).

Let's consider the following uninspired approach for validating the length of a book ISBN:

// Avoid  
public boolean validateIsbnLength(Book book) {  
  
 Optional<String> isbn = book.getIsbn();  
  
 if (isbn.isPresent()) {  
 return isbn.get().length() > 10;  
 }  
  
 return false;  
}Copy

The preceding solution relies on explicitly unwrapping the Optional value. But if we rely on Optional.filter(), we can do it without this explicit unwrapping, as follows:

// Prefer  
public boolean validateIsbnLength(Book book) {  
  
 Optional<String> isbn = book.getIsbn();  
  
 return isbn.filter((i) -> i.length() > 10)  
 .isPresent();  
}Copy

Optional.filter() is also useful for avoiding breaking lambda chains.

247. Chaining the Optional and Stream APIs

Starting with JDK 9, we can refer to an Optional instance as Stream by applying the Optional.stream() method.

This is quite useful when we have to chain the Optional and Stream APIs. The Optional.stream() method returns a Stream of one element (the value of Optional) or an empty Stream (if Optional has no value). Furthermore, we can use all of the methods that are available in the Stream API.

Let's assume that we have a method for fetching books by ISBN (if no book matches the given ISBN, then this method returns an empty Optional object):

public Optional<Book> fetchBookByIsbn(String isbn) {  
 // fetching book by the given "isbn" can return null  
 Book book = ...;  
  
 return Optional.ofNullable(book);  
}Copy

In addition to this, we loop a List of ISBNs and return List of Book as follows (each ISBN is passed through the fetchBookByIsbn() method):

// Avoid  
public List<Book> fetchBooks(List<String> isbns) {  
  
 return isbns.stream()  
 .map(this::fetchBookByIsbn)  
 .filter(Optional::isPresent)  
 .map(Optional::get)  
 .collect(toList());  
}Copy

The focus here is on the following two lines of code:

.filter(Optional::isPresent)  
.map(Optional::get)Copy

Since the fetchBookByIsbn() method can return empty Optional classes, we must ensure that we eliminate them from the final result. For this, we call Stream.filter() and apply the Optional.isPresent() function to each Optional object returned by fetchBookByIsbn(). So, after filtering, we have only Optional classes with present values. Furthermore, we apply the Stream.map() method for unwrapping these Optional classes to Book. Finally, we collect the Book objects in List.

But we can accomplish the same thing more elegantly using Optional.stream(), as follows:

// Prefer  
public List<Book> fetchBooksPrefer(List<String> isbns) {  
  
 return isbns.stream()  
 .map(this::fetchBookByIsbn)  
 .flatMap(Optional::stream)  
 .collect(toList());  
}Copy

Practically, in cases like these, we can use Optional.stream() to replace filter() and map() with flatMap().

Calling Optional.stream() for each Optional<Book> returned by fetchBookByIsbn() will result in Stream<Book> containing a single Book object or nothing (an empty stream). If Optional<Book> doesn't contain a value (is empty), then Stream<Book> is also empty. Relying on flatMap() instead of map() will avoid a result of the Stream<Stream<Book>> type.

As a bonus, we can convert Optional into List as follows:

public static<T> List<T> optionalToList(Optional<T> optional) {  
 return optional.stream().collect(toList());  
}Copy

248. Optional and identity-sensitive operations

*Identity-sensitive* operations include reference equality (==), identity hash-based, or synchronization.

The Optional class is a *value-based* class such as LocalDateTime, therefore identity-sensitive operations should be avoided.

For example, let's test the equality of two Optional classes via ==:

Book book = new Book();  
Optional<Book> op1 = Optional.of(book);  
Optional<Book> op2 = Optional.of(book);  
  
// Avoid  
// op1 == op2 => false, expected true  
if (op1 == op2) {  
 System.out.println("op1 is equal with op2, (via ==)");  
} else {  
 System.out.println("op1 is not equal with op2, (via ==)");  
}Copy

This will give the following output:

op1 is not equal with op2, (via ==)Copy

Since op1 and op2 are not references to the same object, they are not equal so don't conform with the== implementation.

To compare the values, we need to rely on equals(), as follows:

// Prefer  
if (op1.equals(op2)) {  
 System.out.println("op1 is equal with op2, (via equals())");  
} else {  
 System.out.println("op1 is not equal with op2, (via equals())");  
}Copy

This will give the following output:

op1 is equal with op2, (via equals())Copy

In the context of the identity-sensitive operations, never do something like this (think that Optional is a value-based class and such classes should not be used for locking—for more details, see <https://rules.sonarsource.com/java/tag/java8/RSPEC-3436>):

Optional<Book> book = Optional.of(new Book());  
synchronized(book) {  
 ...  
}Copy

249. Returning a boolean if the Optional class is empty

Let's assume that we have the following simple method:

public static Optional<Cart> fetchCart(long userId) {  
 // the shopping cart of the given "userId" can be null  
 Cart cart = ...;  
  
 return Optional.ofNullable(cart);  
}Copy

Now, we want to write a method named cartIsEmpty() that calls the fetchCart() method and returns a flag that is true if the fetched cart is empty. Before JDK 11, we could implement this method based on Optional.isPresent(), as follows:

// Avoid (after JDK 11)  
public static boolean cartIsEmpty(long id) {  
 Optional<Cart> cart = fetchCart(id);  
  
 return !cart.isPresent();  
}Copy

This solution works fine but is not very expressive. We check for emptiness via presence, and we have to negate the isPresent() result.

Since JDK 11, the Optional class has been enriched with a new method named isEmpty(). As its name suggests, this is a flag method that returns true if the tested Optional class is empty. So, we can increase the expressiveness of our solution as follows:

// Prefer (after JDK 11)  
public static boolean cartIsEmpty(long id) {  
 Optional<Cart> cart = fetchCart(id);  
  
 return cart.isEmpty();  
}Copy

Summary

Done! This was the last problem of this chapter. At this point, you should have all of the arguments needed for using Optional correctly.

Download the applications from this chapter to see the results and to see the additional details.

The HTTP Client and WebSocket APIs

This chapter includes 20 problems that are meant to cover the HTTP Client and WebSocket APIs.

Do you remember HttpUrlConnection? Well, JDK 11 comes with the HTTP Client API as a reinvention of HttpUrlConnection. The HTTP Client API is easy to use and supports HTTP/2 (default) and HTTP/1.1. For backward compatibility, the HTTP Client API will automatically downgrade from HTTP/2 to HTTP 1.1 when the server doesn't support HTTP/2. Moreover, the HTTP Client API supports synchronous and asynchronous programming models and relies on streams to transfer data (reactive streams). It also supports the WebSocket protocol, which is used in real-time web applications to provide client-server communication with low message overhead.

Problems

Use the following problems to test your HTTP Client and WebSocket API programming prowess. I strongly encourage you to give each problem a try before you turn to the solutions and download the example programs:

1. **HTTP/2**: Provide a brief overview of the HTTP/2 protocol
2. **Triggering an asynchronous GET request**: Write a program that uses the HTTP Client API to trigger an asynchronous GET request and display the response code and body.
3. **Setting a proxy**: Write a program that uses the HTTP Client API to set up a connection via a proxy.
4. **Setting/getting headers**: Write a program that adds additional headers to the request and gets the headers of the response.
5. **Specifying the HTTP method**: Write a program that specifies the HTTP method of a request (for example, GET, POST, PUT, and DELETE).
6. **Setting the request body**: Write a program that uses the HTTP Client API to add a body to a request.
7. **Setting connection authentication**: Write a program that uses the HTTP Client API to set up a connection authentication via username and password.
8. **Setting a timeout**: Write a program that uses the HTTP Client API to set the amount of time we want to wait for a response (timeout).
9. **Setting the redirect policy**: Write a program that uses the HTTP Client API to automatically redirect if needed.
10. **Sending sync and async requests**: Write a program that sends the same request in sync and async modes.
11. **Handling cookies**: Write a program that uses the HTTP Client API to set a cookie handler.
12. **Getting response information**: Write a program that uses the HTTP Client API to get information about the response (for example, URI, version, headers, status code, body, and so on).
13. **Handling response body types**: Write several snippets of code to exemplify how to handle common response body types via HttpResponse.BodyHandlers.
14. **Getting, updating, and saving a JSON**: Write a program that uses the HTTP Client API to get, update, and save a JSON.
15. **Compression**: Write a program that handles compressed responses (for example, .gzip).
16. **Handling form data**: Write a program that uses the HTTP Client API to submit a data form (application/x-www-form-urlencoded).
17. **Downloading a resource**: Write a program that uses the HTTP Client API to download a resource.
18. **Uploading with multipart**: Write a program that uses the HTTP Client API to upload a resource.
19. **HTTP/2 server push**: Write a program that exemplifies the HTTP/2 server push feature via the HTTP Client API.
20. **WebSocket**: Write a program that opens a connection to a WebSocket endpoint, collects data for 10 seconds, and closes the connection.

Solutions

The following sections describe the solutions to the preceding problems. Remember that there usually isn't a single correct way to solve a particular problem. Also, remember that the explanations that are shown here only include the most interesting and important details that are needed to solve the problems. You can download the example solutions to view additional details and experiment with the programs from <https://github.com/PacktPublishing/Java-Coding-Problems>.

250. HTTP/2

**HTTP/2** is an efficient protocol that substantially and measurably improves the **HTTP/1.1** protocol.

As part of a bigger picture, **HTTP/2** has two parts:

* **The framing layer**: This is the **HTTP/2** multiplexing core ability
* **The data layer**: This contains the data (what we typically refer to as HTTP)

The following diagram depicts the communication in **HTTP/1.1** (top) and **HTTP/2** (bottom):

Interfaz de usuario gráfica

Descripción generada automáticamente

**HTTP/2** is widely adopted by servers and browsers, and it comes with the following improvements over **HTTP/1.1**:

* **Binary protocol**: Less readable by humans but more machine friendly, the **HTTP/2** *framing layer* is a binary framed protocol.
* **Multiplexing**: This refers to interwoven requests and responses. Multiple requests run at the same time on the same connection.
* **Server push**: The server can decide to send additional resources to the client.
* **Single connection to server**: **HTTP/2** uses a single communication line (TCP connection) per origin (domain).
* **Header compression**: **HTTP/2** relies on HPACK compression to reduce headers. This has a significant impact on redundant bytes.
* **Encrypted**: Most of the data that's transferred over the wires is encrypted.

251. Triggering an asynchronous GET request

Triggering asynchronous GET request is a three-step job, as follows:

1. Create a new HttpClient object (java.net.http.HttpClient):

HttpClient client = HttpClient.newHttpClient();Copy

1. Build an HttpRequest object (java.net.http.HttpRequest) and specify the request (by default, this is a GET request):

HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();Copy

For setting the URI, we can call the HttpRequest.newBuilder(URI) constructor or call the uri(URI) method on the Builder instance (like we did previously).

1. Trigger the request and wait for the response (java.net.http.HttpResponse). Being a synchronous request, the application will block until the response is available:

HttpResponse<String> response   
 = client.send(request, BodyHandlers.ofString());Copy

If we group these three steps and add the lines for displaying the response code and body at the console, then we obtain the following code:

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();  
  
HttpResponse<String> response   
 = client.send(request, BodyHandlers.ofString());  
  
System.out.println("Status code: " + response.statusCode());  
System.out.println("\n Body: " + response.body());Copy

One possible output for the preceding code is as follows:

Status code: 200  
Body:  
{  
 "data": {  
 "id": 2,  
 "email": "janet.weaver@reqres.in",  
 "first\_name": "Janet",  
 "last\_name": "Weaver",  
 "avatar": "https://s3.amazonaws.com/..."  
 }  
}Copy

By default, this request takes place using HTTP/2. However, we can explicitly set the version via HttpRequest.Builder.version() as well. This method gets an argument of the HttpClient.Version type, which is an enum data type that exposes two constants: HTTP\_2 and HTTP\_1\_1. The following is an example of explicitly downgrading to HTTP/1.1:

HttpRequest request = HttpRequest.newBuilder()  
 .version(HttpClient.Version.HTTP\_1\_1)  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();Copy

The default settings for HttpClient are as follows:

* HTTP/2
* No authenticator
* No connection timeout
* No cookie handler
* Default thread pool executor
* Redirection policy of NEVER
* Default proxy selector
* Default SSL context

We'll take a look at the query parameter builder in the next section.

Query parameter builder

Working with URIs that contain query parameters implies encoding these parameters. The Java built-in method for accomplishing this task is URLEncoder.encode(). But concatenating and encoding several query parameters leads to something similar to the following:

URI uri = URI.create("http://localhost:8080/books?name=" +  
 URLEncoder.encode("Games & Fun!", StandardCharsets.UTF\_8) +  
 "&no=" + URLEncoder.encode("124#442#000", StandardCharsets.UTF\_8) +  
 "&price=" + URLEncoder.encode("$23.99", StandardCharsets.UTF\_8)  
);Copy

When we have to work with a significant number of query parameters, this solution is not very convenient. We can, however, try to write a helper method to hide the URLEncoder.encode() method in a loop over a collection of query parameters, or we can rely on a URI builder.

In Spring, the URI builder is org.springframework.web.util.UriComponentsBuilder. The following code is self-explanatory:

URI uri = UriComponentsBuilder.newInstance()  
 .scheme("http")  
 .host("localhost")  
 .port(8080)  
 .path("books")  
 .queryParam("name", "Games & Fun!")  
 .queryParam("no", "124#442#000")  
 .queryParam("price", "$23.99")  
 .build()  
 .toUri();Copy

In a non-Spring application, we can rely on a URI builder such as the urlbuilder library (<https://github.com/mikaelhg/urlbuilder>). The code that's bundled with this book contains an example of using this.

252. Setting a proxy

To set up a proxy, we rely on the HttpClient.proxy() method of a Builder method. The proxy() method gets an argument of the ProxySelector type, which can be the system-wide proxy selector (via getDefault()) or the proxy selector that's pointed to via its address (via InetSocketAddress).

Let's assume that we have a proxy at the proxy.host:80 address. We can set up this proxy as follows:

HttpClient client = HttpClient.newBuilder()  
 .proxy(ProxySelector.of(new InetSocketAddress("proxy.host", 80)))  
 .build();Copy

Alternatively, we can set up the system-wide proxy selector, as follows:

HttpClient client = HttpClient.newBuilder()  
 .proxy(ProxySelector.getDefault())  
 .build();Copy

253. Setting/getting headers

HttpRequest and HttpResponse expose a suite of methods for working with headers. We'll learn about these methods in the upcoming sections.

Setting request headers

The HttpRequest.Builder class uses three methods to set additional headers:

* header​(String name, String value) and setHeader​(String name, String value): These are used to add headers one by one, as shown in the following code:

HttpRequest request = HttpRequest.newBuilder()  
 .uri(...)  
 ...  
 .header("key\_1", "value\_1")  
 .header("key\_2", "value\_2")  
 ...  
 .build();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .uri(...)  
 ...  
 .setHeader("key\_1", "value\_1")  
 .setHeader("key\_2", "value\_2")  
 ...  
 .build();Copy

The difference between header() and setHeader() is that the former adds the specified header while the latter sets the specified header. In other words, header() adds the given value to the list of values for that name/key, while setHeader() overwrites any previously set values for that name/key.

* headers​(String... headers): This is used to add headers separated by a comma, as shown in the following code:

HttpRequest request = HttpRequest.newBuilder()  
 .uri(...)  
 ...  
 .headers("key\_1", "value\_1", "key\_2",  
 "value\_2", "key\_3", "value\_3", ...)  
 ...  
 .build();Copy

For example, the Content-Type: application/json and Referer: https://reqres.in/ headers can be added to the request that's triggered by the https://reqres.in/api/users/2 URI, as follows:

HttpRequest request = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .header("Referer", "https://reqres.in/")  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();Copy

You can also do the following:

HttpRequest request = HttpRequest.newBuilder()  
 .setHeader("Content-Type", "application/json")  
 .setHeader("Referer", "https://reqres.in/")  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();Copy

Finally, you can do something like this:

HttpRequest request = HttpRequest.newBuilder()  
 .headers("Content-Type", "application/json",  
 "Referer", "https://reqres.in/")  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();Copy

Depending on the goal, all three methods can be combined in order to specify the request headers.

Getting request/response headers

Getting the request headers can be done using the HttpRequest.headers() method. A similar method exists in HttpResponse for getting the headers of the response. Both methods return an HttpHeaders object.

Both of these methods can be used in the same way, so let's focus on getting the response headers. We can get the headers like so:

HttpResponse<...> response ...  
HttpHeaders allHeaders = response.headers();Copy

Getting all of the values of a header can be done using HttpHeaders.allValues(), as follows:

List<String> allValuesOfCacheControl  
 = response.headers().allValues("Cache-Control");Copy

Getting only the first value of a header can be done using HttpHeaders.firstValue(), as follows:

Optional<String> firstValueOfCacheControl  
 = response.headers().firstValue("Cache-Control");Copy

If the returned value of a header is Long, then rely on HttpHeaders.firstValueAsLong(). This method gets an argument representing the name of the header and returns Optional<Long>. If the value of the specified header cannot be parsed as Long, then NumberFormatException will be thrown.

254. Specifying the HTTP method

We can indicate the HTTP method that's used by our request using the following methods from HttpRequest.Builder:

* GET(): This method sends the request using the HTTP GET method, as shown in the following example:

HttpRequest requestGet = HttpRequest.newBuilder()  
 .GET() // can be omitted since it is default  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();Copy

* POST(): This method sends the request using the HTTP POST method, as shown in the following example:

HttpRequest requestPost = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .POST(HttpRequest.BodyPublishers.ofString(  
 "{\"name\": \"morpheus\",\"job\": \"leader\"}"))  
 .uri(URI.create("https://reqres.in/api/users"))  
 .build();Copy

* PUT(): This method sends the request using the HTTP PUT method, as shown in the following example:

HttpRequest requestPut = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .PUT(HttpRequest.BodyPublishers.ofString(  
 "{\"name\": \"morpheus\",\"job\": \"zion resident\"}"))  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();Copy

* DELETE(): This method sends the request using the HTTP DELETE method, as shown in the following example:

HttpRequest requestDelete = HttpRequest.newBuilder()  
 .DELETE()  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();Copy

The client can handle all types of HTTP methods, not only the predefined methods (GET, POST, PUT, and DELETE). To create a request with a different HTTP method, we just need to call method().

The following solution triggers an HTTP PATCH request:

HttpRequest requestPatch = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .method("PATCH", HttpRequest.BodyPublishers.ofString(  
 "{\"name\": \"morpheus\",\"job\": \"zion resident\"}"))  
 .uri(URI.create("https://reqres.in/api/users/1"))  
 .build();Copy

When no request body is required, we can rely on BodyPublishers.noBody(). The following solution uses the noBody() method to trigger an HTTP HEAD request:

HttpRequest requestHead = HttpRequest.newBuilder()  
 .method("HEAD", HttpRequest.BodyPublishers.noBody())  
 .uri(URI.create("https://reqres.in/api/users/1"))  
 .build();Copy

In the case of multiple similar requests, we can rely on the copy() method to copy the builder, as shown in the following snippet of code:

HttpRequest.Builder builder = HttpRequest.newBuilder()  
 .uri(URI.create("..."));  
  
HttpRequest request1 = builder.copy().setHeader("...", "...").build();  
HttpRequest request2 = builder.copy().setHeader("...", "...").build();Copy

255. Setting a request body

Setting a request body can be accomplished using HttpRequest.Builder.POST() and HttpRequest.Builder.PUT() or by using method() (for example, method("PATCH", HttpRequest.BodyPublisher)). POST() and PUT() take an argument of the HttpRequest.BodyPublisher type. The API comes with several implementations of this interface (BodyPublisher) in the HttpRequest.BodyPublishers class, as follows:

* BodyPublishers.ofString()
* BodyPublishers.ofFile()
* BodyPublishers.ofByteArray()
* BodyPublishers.ofInputStream()

We'll take a look at these implementations in the following sections.

Creating a body from a string

Creating a body from a string can be accomplished using BodyPublishers.ofString(), as shown in the following snippet of code:

HttpRequest requestBody = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .POST(HttpRequest.BodyPublishers.ofString(  
 "{\"name\": \"morpheus\",\"job\": \"leader\"}"))  
 .uri(URI.create("https://reqres.in/api/users"))  
 .build();Copy

For specifying a charset call, use ofString(String s, Charset charset).

Creating a body from InputStream

Creating a body from InputStream can be accomplished using BodyPublishers.ofInputStream(), as shown in the following snippet of code (here, we rely on ByteArrayInputStream but, of course, any other InputStream is suitable):

HttpRequest requestBodyOfInputStream = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .POST(HttpRequest.BodyPublishers.ofInputStream(()  
 -> inputStream("user.json")))  
 .uri(URI.create("https://reqres.in/api/users"))  
 .build();  
  
private static ByteArrayInputStream inputStream(String fileName) {  
  
 try (ByteArrayInputStream inputStream = new ByteArrayInputStream(  
 Files.readAllBytes(Path.of(fileName)))) {  
  
 return inputStream;  
 } catch (IOException ex) {  
 throw new RuntimeException("File could not be read", ex);  
 }  
}Copy

In order to take advantage of lazy creation, InputStream has to be passed as Supplier.

Creating a body from a byte array

Creating a body from a byte array can be accomplished using BodyPublishers.ofByteArray(), as shown in the following snippet of code:

HttpRequest requestBodyOfByteArray = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .POST(HttpRequest.BodyPublishers.ofByteArray(  
 Files.readAllBytes(Path.of("user.json"))))  
 .uri(URI.create("https://reqres.in/api/users"))  
 .build();Copy

We can also send only a part of the byte array using ofByteArray(byte[] buf, int offset, int length). Moreover, we can provide data from Iterable of byte arrays using ofByteArrays(Iterable<byte[]> iter).

Creating a body from a file

Creating a body from a file can be accomplished using BodyPublishers.ofFile(), as shown in the following snippet of code:

HttpRequest requestBodyOfFile = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .POST(HttpRequest.BodyPublishers.ofFile(Path.of("user.json")))  
 .uri(URI.create("https://reqres.in/api/users"))  
 .build();Copy

256. Setting connection authentication

Typically, authentication to a server is accomplished using a username and password. In code form, this can be done by using the Authenticator class (this negotiates the credentials for HTTP authentication) and the PasswordAuthentication class (the holder for the username and password) together, as follows:

HttpClient client = HttpClient.newBuilder()  
 .authenticator(new Authenticator() {  
  
 @Override  
 protected PasswordAuthentication getPasswordAuthentication() {  
  
 return new PasswordAuthentication(  
 "username",  
 "password".toCharArray());  
 }  
 })  
 .build();Copy

Furthermore, the client can be used to send requests:

HttpRequest request = HttpRequest.newBuilder()  
 ...  
 .build();  
  
HttpResponse<String> response  
 = client.send(request, HttpResponse.BodyHandlers.ofString());Copy

Authenticator supports different authentication schemes (for example, *basic* or *digest* authentication).

Another solution consists of adding credentials in the header, as follows:

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .header("Authorization", basicAuth("username", "password"))  
 ...  
 .build();  
  
HttpResponse<String> response   
 = client.send(request, HttpResponse.BodyHandlers.ofString());  
  
private static String basicAuth(String username, String password) {  
 return "Basic " + Base64.getEncoder().encodeToString(  
 (username + ":" + password).getBytes());  
}Copy

In the case of a Bearer authentication (HTTP bearer token), we do the following:

HttpRequest request = HttpRequest.newBuilder()  
 .header("Authorization",   
 "Bearer mT8JNMyWCG0D7waCHkyxo0Hm80YBqelv5SBL")  
 .uri(URI.create("https://gorest.co.in/public-api/users"))  
 .build();Copy

We can also do this in the body of a POST request:

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .POST(BodyPublishers.ofString("{\"email\":\"eve.holt@reqres.in\",  
 \"password\":\"cityslicka\"}"))  
 .uri(URI.create("https://reqres.in/api/login"))  
 .build();  
  
HttpResponse<String> response   
 = client.send(request, HttpResponse.BodyHandlers.ofString());Copy

Different requests can use different credentials. Moreover, Authenticator provides a suite of methods (for example, getRequestingSite()) that are useful if we wish to find out what values should be provided. In production, the application should not provide the credentials in plaintext, like they were in these examples.

257. Setting a timeout

By default, a request has no timeout (infinite timeout). To set the amount of time we want to wait for a response (timeout), we can call the HttpRequest.Builder.timeout() method. This method gets an argument of the Duration type, which can be used like so:

HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .timeout(Duration.of(5, ChronoUnit.MILLIS))  
 .build();Copy

If the specified timeout elapses, then java.net.http.HttpConnectTimeoutException will be thrown.

258. Setting the redirect policy

When we try to access a resource that was moved to a different URI, the server will return an HTTP status code in the range of 3xx, as well as information about the new URI. Browsers are capable of automatically sending another request to the new location when they receive a redirect response (301, 302, 303, 307, and 308).

The HTTP Client API can automatically redirect to this new URI if we explicitly set the redirect policy via followRedirects(), as follows:

HttpClient client = HttpClient.newBuilder()  
 .followRedirects(HttpClient.Redirect.ALWAYS)  
 .build();Copy

To never redirect, just give the HttpClient.Redirect.NEVER constant to followRedirects() (this is the default).

To always redirect, except from HTTPS URLs to HTTP URLs, just give the HttpClient.Redirect.NORMAL constant to followRedirects().

When the redirect policy is not set to ALWAYS, the application is responsible for handling redirects. Commonly, this is accomplished by reading the new address from the HTTP Location header, as follows (the following code is only interested in redirecting if the returned status code is 301 (moved permanently) or 308 (permanent redirect)):

int sc = response.statusCode();  
  
if (sc == 301 || sc == 308) { // use an enum for HTTP response codes  
 String newLocation = response.headers()  
 .firstValue("Location").orElse("");  
  
 // handle the redirection to newLocation  
}Copy

A redirect can be easily detected by comparing the request URI with the response URI. If they are not the same, then a redirect occurs:

if (!request.uri().equals(response.uri())) {  
 System.out.println("The request was redirected to: "   
 + response.uri());  
}Copy

259. Sending sync and async requests

Sending a request to a server can be accomplished using the following two methods from HttpClient:

* send(): This method sends a request synchronously (this will block until the response is available or a timeout occurs)
* sendAsync(): This method sends a request asynchronously (non-blocking)

We'll explain the different ways we can send a request in the next section.

Sending a request synchronously

We've already done this in the previous problems, and so we will just provide you with a quick remainder, as follows:

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();  
  
HttpResponse<String> response   
 = client.send(request, HttpResponse.BodyHandlers.ofString());Copy

Sending a request asynchronously

In order to send requests asynchronously, the HTTP Client API relies on CompletableFeature, as discussed in [Chapter 11](https://subscription.packtpub.com/book/programming/9781789801415/13/ch13lvl1sec52/b466dd92-c030-4835-b259-fdf24f4a36b1.xhtml), *Concurrency – Deep Dive*, and the sendAsync() method, as follows:

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();  
  
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())  
 .thenApply(HttpResponse::body)  
 .exceptionally(e -> "Exception: " + e)  
 .thenAccept(System.out::println)  
 .get(30, TimeUnit.SECONDS); // or join()Copy

Alternatively, let's assume that, while waiting for the response, we want to execute other tasks as well:

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();  
  
CompletableFuture<String> response  
 = client.sendAsync(request, HttpResponse.BodyHandlers.ofString())  
 .thenApply(HttpResponse::body)  
 .exceptionally(e -> "Exception: " + e);  
  
while (!response.isDone()) {  
 Thread.sleep(50);  
 System.out.println("Perform other tasks   
 while waiting for the response ...");  
}  
  
String body = response.get(30, TimeUnit.SECONDS); // or join()  
System.out.println("Body: " + body);Copy

Sending multiple requests concurrently

How do we send multiple requests concurrently and wait for all of the responses to be available?

As we know, CompletableFuture comes with the allOf() method (for more details, please read [Chapter 11](https://subscription.packtpub.com/book/programming/9781789801415/13/ch13lvl1sec52/b466dd92-c030-4835-b259-fdf24f4a36b1.xhtml), *Concurrency – Deep Dive*), which can execute tasks in parallel and waits for all of them to complete. CompletableFuture<Void> is returned.

The following code waits for the responses to four requests:

List<URI> uris = Arrays.asList(  
 new URI("https://reqres.in/api/users/2"), // one user  
 new URI("https://reqres.in/api/users?page=2"), // list of users  
 new URI("https://reqres.in/api/unknown/2"), // list of resources  
 new URI("https://reqres.in/api/users/23")); // user not found  
  
HttpClient client = HttpClient.newHttpClient();  
  
List<HttpRequest> requests = uris.stream()  
 .map(HttpRequest::newBuilder)  
 .map(reqBuilder -> reqBuilder.build())  
 .collect(Collectors.toList());  
  
CompletableFuture.allOf(requests.stream()  
 .map(req -> client.sendAsync(  
 req, HttpResponse.BodyHandlers.ofString())  
 .thenApply((res) -> res.uri() + " | " + res.body() + "\n")  
 .exceptionally(e -> "Exception: " + e)  
 .thenAccept(System.out::println))  
 .toArray(CompletableFuture<?>[]::new))  
 .join();Copy

To collect the bodies of the responses (for example, in List<String>), consider the WaitAllResponsesFetchBodiesInList class, which is available in the code that's bundled with this book.

Using a custom Executor object can be accomplished as follows:

ExecutorService executor = Executors.newFixedThreadPool(5);  
  
HttpClient client = HttpClient.newBuilder()  
 .executor(executor)  
 .build();Copy

260. Handling cookies

By default, JDK 11's HTTP Client supports cookies, but there are instances where built-in support is disabled. We can enable it as follows:

HttpClient client = HttpClient.newBuilder()  
 .cookieHandler(new CookieManager())  
 .build();Copy

So, the HTTP Client API allows us to set a cookie handler using the HttpClient.Builder.cookieHandler() method. This method gets an argument of the CookieManager type.

The following solution sets CookieManager that doesn't accept cookies:

HttpClient client = HttpClient.newBuilder()  
 .cookieHandler(new CookieManager(null, CookiePolicy.ACCEPT\_NONE))  
 .build();Copy

For accepting cookies, set CookiePolicy to ALL (accept all cookies) or ACCEPT\_ORIGINAL\_SERVER (accept cookies only from the original server).

The following solutions accept all cookies and display them in the console (if any credentials are reported as invalid, then consider obtaining a new token from <https://gorest.co.in/rest-console.html>):

CookieManager cm = new CookieManager();  
cm.setCookiePolicy(CookiePolicy.ACCEPT\_ALL);  
  
HttpClient client = HttpClient.newBuilder()  
 .cookieHandler(cm)  
 .build();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .header("Authorization",   
 "Bearer mT8JNMyWCG0D7waCHkyxo0Hm80YBqelv5SBL")  
 .uri(URI.create("https://gorest.co.in/public-api/users/1"))  
 .build();  
  
HttpResponse<String> response  
 = client.send(request, HttpResponse.BodyHandlers.ofString());  
  
System.out.println("Status code: " + response.statusCode());  
System.out.println("\n Body: " + response.body());  
  
CookieStore cookieStore = cm.getCookieStore();  
System.out.println("\nCookies: " + cookieStore.getCookies());Copy

Checking the set-cookie header can be done as follows:

Optional<String> setcookie   
 = response.headers().firstValue("set-cookie");Copy

261. Getting response information

In order to get information about the response, we can rely on the methods from the HttpResponse class. The names of these methods are very intuitive; therefore, the following snippet of code is self-explanatory:

...  
HttpResponse<String> response  
 = client.send(request, HttpResponse.BodyHandlers.ofString());  
  
System.out.println("Version: " + response.version());  
System.out.println("\nURI: " + response.uri());  
System.out.println("\nStatus code: " + response.statusCode());  
System.out.println("\nHeaders: " + response.headers());  
System.out.println("\n Body: " + response.body());Copy

Consider exploring the documentation to find more useful methods.

262. Handling response body types

Handling response body types can be accomplished using HttpResponse.BodyHandler. The API comes with several implementations of this interface (BodyHandler) in the HttpResponse.BodyHandlers class, as follows:

* BodyHandlers.ofByteArray()
* BodyHandlers.ofFile()
* BodyHandlers.ofString()
* BodyHandlers.ofInputStream()
* BodyHandlers.ofLines()

Considering the following request, let's look at several solutions for handling the response body:

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();Copy

We'll look at how to handle different types of response bodies in the following sections.

Handling a response body as a string

Handling a body response as a string can be accomplished using BodyHandlers.ofString(), as shown in the following snippet of code:

HttpResponse<String> responseOfString  
 = client.send(request, HttpResponse.BodyHandlers.ofString());  
  
System.out.println("Status code: " + responseOfString.statusCode());  
System.out.println("Body: " + responseOfString.body());Copy

For specifying a charset, call ofString(String s, Charset charset).

Handling a response body as a file

Handling a body response as a file can be accomplished using BodyHandlers.ofFile(), as shown in the following snippet of code:

HttpResponse<Path> responseOfFile = client.send(  
 request, HttpResponse.BodyHandlers.ofFile(  
 Path.of("response.json")));  
  
System.out.println("Status code: " + responseOfFile.statusCode());  
System.out.println("Body: " + responseOfFile.body());Copy

For specifying the open options, call ofFile(Path file, OpenOption... openOptions).

Handling a response body as a byte array

Handling a body response as a byte array can be accomplished using BodyHandlers.ofByteArray(), as shown in the following snippet of code:

HttpResponse<byte[]> responseOfByteArray = client.send(  
 request, HttpResponse.BodyHandlers.ofByteArray());  
  
System.out.println("Status code: "   
 + responseOfByteArray.statusCode());  
System.out.println("Body: "  
 + new String(responseOfByteArray.body()));Copy

For consuming the byte array, call ofByteArrayConsumer(Consumer<Optional<byte[]>> consumer).

Handling a response body as an input stream

Handling a body response as InputStream can be accomplished using BodyHandlers.ofInputStream(), as shown in the following snippet of code:

HttpResponse<InputStream> responseOfInputStream = client.send(  
 request, HttpResponse.BodyHandlers.ofInputStream());  
  
System.out.println("\nHttpResponse.BodyHandlers.ofInputStream():");  
System.out.println("Status code: "   
 + responseOfInputStream.statusCode());  
  
byte[] allBytes;  
try (InputStream fromIs = responseOfInputStream.body()) {  
 allBytes = fromIs.readAllBytes();  
}  
  
System.out.println("Body: "  
 + new String(allBytes, StandardCharsets.UTF\_8));Copy

Handling a response body as a stream of strings

Handling a body response as a stream of strings can be accomplished using BodyHandlers.ofLines(), as shown in the following snippet of code:

HttpResponse<Stream<String>> responseOfLines = client.send(  
 request, HttpResponse.BodyHandlers.ofLines());  
  
System.out.println("Status code: " + responseOfLines.statusCode());  
System.out.println("Body: "   
 + responseOfLines.body().collect(toList()));Copy

263. Getting, updating, and saving a JSON

In the previous problems, we manipulated JSON data as plaintext (strings). The HTTP Client API doesn't provide special or dedicated support for JSON data and treats this kind of data as any other string.

Nevertheless, we are used to representing JSON data as Java objects (POJOs) and relying on the conversion between JSON and Java when needed. We can write a solution to our problem without involving the HTTP Client API. However, we can also write a solution using a custom implementation of HttpResponse.BodyHandler that relies on a JSON parser to convert the response into Java objects. For example, we can rely on JSON-B (introduced in [Chapter 6](https://subscription.packtpub.com/book/programming/9781789801415/13/ch13lvl1sec52/d238667b-fc00-4e12-a82e-aaa18633e9d9.xhtml), *Java I/O Paths, Files, Buffers, Scanning, and Formatting*).

Implementing the HttpResponse.BodyHandler interface implies overriding the apply(HttpResponse.ResponseInfo responseInfo) method. Using this method, we can take the bytes from the response and convert them into a Java object. The code is as follows:

public class JsonBodyHandler<T>   
 implements HttpResponse.BodyHandler<T> {  
  
 private final Jsonb jsonb;  
 private final Class<T> type;  
  
 private JsonBodyHandler(Jsonb jsonb, Class<T> type) {  
 this.jsonb = jsonb;  
 this.type = type;  
 }  
  
 public static <T> JsonBodyHandler<T>   
 jsonBodyHandler(Class<T> type) {  
 return jsonBodyHandler(JsonbBuilder.create(), type);  
 }  
  
 public static <T> JsonBodyHandler<T> jsonBodyHandler(  
 Jsonb jsonb, Class<T> type) {  
 return new JsonBodyHandler<>(jsonb, type);  
 }  
  
 @Override  
 public HttpResponse.BodySubscriber<T> apply(  
 HttpResponse.ResponseInfo responseInfo) {  
  
 return BodySubscribers.mapping(BodySubscribers.ofByteArray(),  
 byteArray -> this.jsonb.fromJson(  
 new ByteArrayInputStream(byteArray), this.type));  
 }  
}Copy

Let's assume that the JSON that we want to manipulate looks like the following (this is the response from the server):

{  
 "data": {  
 "id": 2,  
 "email": "janet.weaver@reqres.in",  
 "first\_name": "Janet",  
 "last\_name": "Weaver",  
 "avatar": "https://s3.amazonaws.com/..."  
 }  
}Copy

The Java objects for representing this JSON are as follows:

public class User {  
  
 private Data data;  
 private String updatedAt;  
  
 // getters, setters and toString()  
}  
  
public class Data {  
  
 private Integer id;  
 private String email;  
  
 @JsonbProperty("first\_name")  
 private String firstName;  
  
 @JsonbProperty("last\_name")  
 private String lastName;  
  
 private String avatar;  
  
 // getters, setters and toString()  
}Copy

Now, let's see how we can manipulate the JSON in requests and responses.

JSON response to User

The following solution triggers a GET request and converts the returned JSON response into User:

Jsonb jsonb = JsonbBuilder.create();  
HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest requestGet = HttpRequest.newBuilder()  
 .uri(URI.create("https://reqres.in/api/users/2"))  
 .build();  
  
HttpResponse<User> responseGet = client.send(  
 requestGet, JsonBodyHandler.jsonBodyHandler(jsonb, User.class));  
  
User user = responseGet.body();Copy

Updated User to JSON request

The following solution updates the email address of the user we fetched in the preceding subsection:

user.getData().setEmail("newemail@gmail.com");  
  
HttpRequest requestPut = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .uri(URI.create("https://reqres.in/api/users"))  
 .PUT(HttpRequest.BodyPublishers.ofString(jsonb.toJson(user)))  
 .build();  
  
HttpResponse<User> responsePut = client.send(  
 requestPut, JsonBodyHandler.jsonBodyHandler(jsonb, User.class));  
  
User updatedUser = responsePut.body();Copy

New User to JSON request

The following solution creates a new user (the response status code should be 201):

Data data = new Data();  
data.setId(10);  
data.setFirstName("John");  
data.setLastName("Year");  
data.setAvatar("https://johnyear.com/jy.png");  
  
User newUser = new User();  
newUser.setData(data);  
  
HttpRequest requestPost = HttpRequest.newBuilder()  
 .header("Content-Type", "application/json")  
 .uri(URI.create("https://reqres.in/api/users"))  
 .POST(HttpRequest.BodyPublishers.ofString(jsonb.toJson(user)))  
 .build();  
  
HttpResponse<Void> responsePost = client.send(  
 requestPost, HttpResponse.BodyHandlers.discarding());  
  
int sc = responsePost.statusCode(); // 201Copy

Note that we ignore any response body via HttpResponse.BodyHandlers.discarding().

264. Compression

Enabling .gzip compression on the server is a common practice that's meant to significantly improve the site's load time. But JDK 11's HTTP Client API doesn't take advantage of .gzip compression. In other words, the HTTP Client API doesn't require compressed responses and doesn't know how to deal with such responses.

To request compressed responses, we have to send the Accept-Encoding header with the .gzip value. This header is not added by the HTTP Client API, so we will add it as follows:

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .header("Accept-Encoding", "gzip")  
 .uri(URI.create("https://davidwalsh.name"))  
 .build();Copy

This is just half of the job. So far, if the gzip encoding is enabled on the server, then we will receive a compressed response. To detect whether the response is compressed or not, we have to check the Encoding header, as follows:

HttpResponse<InputStream> response = client.send(  
 request, HttpResponse.BodyHandlers.ofInputStream());  
  
String encoding = response.headers()  
 .firstValue("Content-Encoding").orElse("");  
  
if ("gzip".equals(encoding)) {  
 String gzipAsString = gZipToString(response.body());  
 System.out.println(gzipAsString);  
} else {  
 String isAsString = isToString(response.body());  
 System.out.println(isAsString);  
}Copy

The gZipToString() method is a helper method that takes InputStream and treats it as GZIPInputStream. In other words, this method reads the bytes from the given input stream and uses them to create a string:

public static String gzipToString(InputStream gzip)   
 throws IOException {  
  
 byte[] allBytes;  
 try (InputStream fromIs = new GZIPInputStream(gzip)) {  
 allBytes = fromIs.readAllBytes();  
 }  
  
 return new String(allBytes, StandardCharsets.UTF\_8);  
}Copy

If the response is not compressed, then isToString() is the helper method that we need:

public static String isToString(InputStream is) throws IOException {  
  
 byte[] allBytes;  
 try (InputStream fromIs = is) {  
 allBytes = fromIs.readAllBytes();  
 }  
  
 return new String(allBytes, StandardCharsets.UTF\_8);  
}Copy

265. Handling form data

JDK 11's HTTP Client API doesn't come with built-in support for triggering POST requests with x-www-form-urlencoded. The solution to this problem is to rely on a custom BodyPublisher class.

Writing a custom BodyPublisher class is pretty simple if we consider the following:

* Data is represented as key-value pairs
* Each pair is a key = value type
* Pairs are separated via the & character
* Keys and values should be properly encoded

Since data is represented as key-value pairs, it's very convenient to store in Map. Furthermore, we just loop this Map and apply the preceding information, as follows:

public class FormBodyPublisher {  
  
 public static HttpRequest.BodyPublisher ofForm(  
 Map<Object, Object> data) {  
  
 StringBuilder body = new StringBuilder();  
  
 for (Object dataKey: data.keySet()) {  
 if (body.length() > 0) {  
 body.append("&");  
 }  
  
 body.append(encode(dataKey))  
 .append("=")  
 .append(encode(data.get(dataKey)));  
 }  
  
 return HttpRequest.BodyPublishers.ofString(body.toString());  
 }  
  
 private static String encode(Object obj) {  
 return URLEncoder.encode(obj.toString(), StandardCharsets.UTF\_8);  
 }  
}Copy

Relying on this solution, a POST (x-www-form-urlencoded) request can be triggered as follows:

Map<Object, Object> data = new HashMap<>();  
data.put("firstname", "John");  
data.put("lastname", "Year");  
data.put("age", 54);  
data.put("avatar", "https://avatars.com/johnyear");  
  
HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .header("Content-Type", "application/x-www-form-urlencoded")  
 .uri(URI.create("http://jkorpela.fi/cgi-bin/echo.cgi"))  
 .POST(FormBodyPublisher.ofForm(data))  
 .build();  
  
HttpResponse<String> response = client.send(  
 request, HttpResponse.BodyHandlers.ofString());Copy

In this case, the response is just an echo of the sent data. Depending on the server's response, the application needs to deal with it, as shown in the *Handling response body types* section.

266. Downloading a resource

As we saw in the *Setting a request body* and *Handling response body types* sections, the HTTP Client API can send and receive text and binary data (for example, images, videos, and so on).

Downloading a file relies on the following two coordinates:

* Sending a GET request
* Handling the received bytes (for example, via BodyHandlers.ofFile())

The following code downloads hibernate-core-5.4.2.Final.jar from the Maven repository in the project classpath:

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("http://.../hibernate-core-5.4.2.Final.jar"))  
 .build();  
  
HttpResponse<Path> response   
 = client.send(request, HttpResponse.BodyHandlers.ofFile(  
 Path.of("hibernate-core-5.4.2.Final.jar")));Copy

If the resource to download is delivered via the Content-Disposition HTTP header, which is of the Content-Disposition attachment; filename="..." type, then we can rely on BodyHandlers.ofFileDownload(), as in the following example:

import static java.nio.file.StandardOpenOption.CREATE;  
...  
HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("http://...downloadfile.php  
 ?file=Hello.txt&cd=attachment+filename"))  
 .build();  
  
HttpResponse<Path> response = client.send(request,  
 HttpResponse.BodyHandlers.ofFileDownload(Path.of(  
 System.getProperty("user.dir")), CREATE));Copy

More files that can be tested are available here: <http://demo.borland.com/testsite/download_testpage.php>.

267. Uploading with multipart

As we saw in the *Setting a request body* section, we can send a file (text or binary) to the server via BodyPublishers.ofFile() and a POST request.

But sending a classical upload request may involve a multipart form POST with Content-Type as multipart/form-data.

In this case, the request body is made of parts that are delimited by a boundary, as shown in the following illustration (--779d334bbfa... is the boundary):

Texto, Carta

Descripción generada automáticamente

However, JDK 11's HTTP Client API doesn't provide built-in support for building this kind of request body. Nevertheless, by following the preceding screenshot, we can define a custom BodyPublisher as follows:

public class MultipartBodyPublisher {  
  
 private static final String LINE\_SEPARATOR = System.lineSeparator();  
  
 public static HttpRequest.BodyPublisher ofMultipart(  
 Map<Object, Object> data, String boundary) throws IOException {  
  
 final byte[] separator = ("--" + boundary +  
 LINE\_SEPARATOR + "Content-Disposition: form-data;  
 name = ").getBytes(StandardCharsets.UTF\_8);  
  
 final List<byte[] > body = new ArrayList<>();  
  
 for (Object dataKey: data.keySet()) {  
  
 body.add(separator);  
 Object dataValue = data.get(dataKey);  
  
 if (dataValue instanceof Path) {  
 Path path = (Path) dataValue;  
 String mimeType = fetchMimeType(path);  
  
 body.add(("\"" + dataKey + "\"; filename=\"" +  
 path.getFileName() + "\"" + LINE\_SEPARATOR +  
 "Content-Type: " + mimeType + LINE\_SEPARATOR +  
 LINE\_SEPARATOR).getBytes(StandardCharsets.UTF\_8));  
  
 body.add(Files.readAllBytes(path));  
 body.add(LINE\_SEPARATOR.getBytes(StandardCharsets.UTF\_8));  
 } else {  
 body.add(("\"" + dataKey + "\"" + LINE\_SEPARATOR +  
 LINE\_SEPARATOR + dataValue + LINE\_SEPARATOR)  
 .getBytes(StandardCharsets.UTF\_8));  
 }  
 }  
  
 body.add(("--" + boundary   
 + "--").getBytes(StandardCharsets.UTF\_8));  
  
 return HttpRequest.BodyPublishers.ofByteArrays(body);  
 }  
  
 private static String fetchMimeType(  
 Path filenamePath) throws IOException {  
  
 String mimeType = Files.probeContentType(filenamePath);  
  
 if (mimeType == null) {  
 throw new IOException("Mime type could not be fetched");  
 }  
  
 return mimeType;  
 }  
 }Copy

Now, we can create a multipart request, as follows (we try to upload a text file called LoremIpsum.txt to a server that simply sent back the raw form data):

Map<Object, Object> data = new LinkedHashMap<>();  
data.put("author", "Lorem Ipsum Generator");  
data.put("filefield", Path.of("LoremIpsum.txt"));  
  
String boundary = UUID.randomUUID().toString().replaceAll("-", "");  
  
HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .header("Content-Type", "multipart/form-data;boundary=" + boundary)  
 .POST(MultipartBodyPublisher.ofMultipart(data, boundary))  
 .uri(URI.create("http://jkorpela.fi/cgi-bin/echoraw.cgi"))  
 .build();  
  
HttpResponse<String> response = client.send(  
 request, HttpResponse.BodyHandlers.ofString());Copy

The response should be similar to the following (the boundary is just a random UUID):

--7ea7a8311ada4804ab11d29bcdedcc55  
Content-Disposition: form-data; name="author"  
Lorem Ipsum Generator  
--7ea7a8311ada4804ab11d29bcdedcc55  
Content-Disposition: form-data; name="filefield"; filename="LoremIpsum.txt"  
Content-Type: text/plain  
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do   
eiusmod tempor incididunt ut labore et dolore magna aliqua.  
--7ea7a8311ada4804ab11d29bcdedcc55--Copy

268. HTTP/2 server push

Besides *multiplexing*, another powerful feature of HTTP/2 is its *server push* capability.

Mainly, in the traditional approach (HTTP/1.1), a browser triggers a request for getting an HTML page and parses the received markup to identify the referenced resources (for example, JS, CSS, images, and so on). To fetch these resources, the browser sends additional requests (one request for each referenced resource). On the other hand, HTTP/2 sends the HTML page and the referenced resources without explicit requests from the browser. So, the browser requests the HTML page and receives the page and everything else that's needed for displaying the page.

The HTTP Client API supports this HTTP/2 feature via the PushPromiseHandler interface. The implementation of this interface must be given as the third argument of the send() or sendAsync() method.

PushPromiseHandler relies on three coordinates, as follows:

* The initiating client send request (initiatingRequest)
* The synthetic push request (pushPromiseRequest)
* The acceptor function, which must be successfully invoked to accept the push promise (acceptor)

A push promise is accepted by invoking the given acceptor function. The acceptor function must be passed a non-null BodyHandler, which is used to handle the promise's response body. The acceptor function will return a CompletableFuture instance that completes the promise's response.

Based on this information, let's look at an implementation of PushPromiseHandler:

private static final List<CompletableFuture<Void>>  
 asyncPushRequests = new CopyOnWriteArrayList<>();  
...  
private static HttpResponse.PushPromiseHandler<String>  
 pushPromiseHandler() {  
  
 return (HttpRequest initiatingRequest,   
 HttpRequest pushPromiseRequest,  
 Function<HttpResponse.BodyHandler<String> ,  
 CompletableFuture<HttpResponse<String>>> acceptor) -> {  
 CompletableFuture<Void> pushcf =  
 acceptor.apply(HttpResponse.BodyHandlers.ofString())  
 .thenApply(HttpResponse::body)  
 .thenAccept((b) -> System.out.println(  
 "\nPushed resource body:\n " + b));  
  
 asyncPushRequests.add(pushcf);  
  
 System.out.println("\nJust got promise push number: " +  
 asyncPushRequests.size());  
 System.out.println("\nInitial push request: " +  
 initiatingRequest.uri());  
 System.out.println("Initial push headers: " +  
 initiatingRequest.headers());  
 System.out.println("Promise push request: " +  
 pushPromiseRequest.uri());  
 System.out.println("Promise push headers: " +  
 pushPromiseRequest.headers());  
 };  
 }Copy

Now, let's trigger a request and pass this PushPromiseHandler to sendAsync():

HttpClient client = HttpClient.newHttpClient();  
  
HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://http2.golang.org/serverpush"))  
 .build();  
  
client.sendAsync(request,  
 HttpResponse.BodyHandlers.ofString(), pushPromiseHandler())  
 .thenApply(HttpResponse::body)  
 .thenAccept((b) -> System.out.println("\nMain resource:\n" + b))  
 .join();  
  
asyncPushRequests.forEach(CompletableFuture::join);  
  
System.out.println("\nFetched a total of " +  
 asyncPushRequests.size() + " push requests");Copy

If we want to return a push promise handler that accumulates push promises, and their responses, into the given map, then we can rely on the PushPromiseHandler.of() method, as follows:

private static final ConcurrentMap<HttpRequest,  
 CompletableFuture<HttpResponse<String>>> promisesMap   
 = new ConcurrentHashMap<>();  
  
private static final Function<HttpRequest,  
 HttpResponse.BodyHandler<String>> promiseHandler   
 = (HttpRequest req) -> HttpResponse.BodyHandlers.ofString();  
  
public static void main(String[] args)  
 throws IOException, InterruptedException {  
  
 HttpClient client = HttpClient.newHttpClient();  
  
 HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://http2.golang.org/serverpush"))  
 .build();  
  
 client.sendAsync(request,  
 HttpResponse.BodyHandlers.ofString(), pushPromiseHandler())  
 .thenApply(HttpResponse::body)  
 .thenAccept((b) -> System.out.println("\nMain resource:\n" + b))  
 .join();  
  
 System.out.println("\nPush promises map size: " +  
 promisesMap.size() + "\n");  
  
 promisesMap.entrySet().forEach((entry) -> {  
 System.out.println("Request = " + entry.getKey() +  
 ", \nResponse = " + entry.getValue().join().body());  
 });  
}  
  
private static HttpResponse.PushPromiseHandler<String>  
 pushPromiseHandler() {  
  
 return HttpResponse.PushPromiseHandler  
 .of(promiseHandler, promisesMap);  
 }Copy

In both solutions of the preceding solutions, we have used a BodyHandler of the String type via ofString(). This is not very useful if the server pushes binary data as well (for example, images). So, if we are dealing with binary data, we need to switch to BodyHandler of the byte[] type via ofByteArray(). Alternatively, we can send the pushed resources to disk via ofFile(), as shown in the following solution, which is an adapted version of the preceding solution:

private static final ConcurrentMap<HttpRequest,  
 CompletableFuture<HttpResponse<Path>>>  
 promisesMap = new ConcurrentHashMap<>();  
  
private static final Function<HttpRequest,  
 HttpResponse.BodyHandler<Path>> promiseHandler   
 = (HttpRequest req) -> HttpResponse.BodyHandlers.ofFile(  
 Paths.get(req.uri().getPath()).getFileName());  
  
public static void main(String[] args)  
 throws IOException, InterruptedException {  
  
 HttpClient client = HttpClient.newHttpClient();  
  
 HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("https://http2.golang.org/serverpush"))  
 .build();  
  
 client.sendAsync(request, HttpResponse.BodyHandlers.ofFile(  
 Path.of("index.html")), pushPromiseHandler())  
 .thenApply(HttpResponse::body)  
 .thenAccept((b) -> System.out.println("\nMain resource:\n" + b))  
 .join();  
  
 System.out.println("\nPush promises map size: " +  
 promisesMap.size() + "\n");  
  
 promisesMap.entrySet().forEach((entry) -> {  
 System.out.println("Request = " + entry.getKey() +  
 ", \nResponse = " + entry.getValue().join().body());  
 });  
}  
  
private static HttpResponse.PushPromiseHandler<Path>  
 pushPromiseHandler() {  
  
 return HttpResponse.PushPromiseHandler  
 .of(promiseHandler, promisesMap);  
 }Copy

The preceding code should save the pushed resources in the application classpath, as shown in the following screenshot:

Texto

Descripción generada automáticamente

269. WebSocket

The HTTP Client supports the WebSocket protocol. In API terms, the core of the implementation is the java.net.http.WebSocket interface. This interface exposes a suite of methods for handling WebSocket communication.

Building a WebSocket instance asynchronously can be accomplished via HttpClient.newWebSocketBuilder().buildAsync().

For example, we can connect to the well known Meetup RSVP WebSocket endpoint (ws://stream.meetup.com/2/rsvps), as follows:

HttpClient client = HttpClient.newHttpClient();  
  
WebSocket webSocket = client.newWebSocketBuilder()  
 .buildAsync(URI.create("ws://stream.meetup.com/2/rsvps"),   
 wsListener).get(10, TimeUnit.SECONDS);Copy

By its nature, the WebSocket protocol is bidirectional. In order to send data, we can rely on sendText(), sendBinary(), sendPing(), and sendPong(). The Meetup RSVP doesn't process the messages that we send but, just for fun, we can send a text message, as follows:

webSocket.sendText("I am an Meetup RSVP fan", true);Copy

The boolean argument is used to mark the end of the message. If this invocation doesn't complete, the message passes false.

To close the connection, we need to use sendClose(), as follows:

webSocket.sendClose(WebSocket.NORMAL\_CLOSURE, "ok");Copy

Finally, we need to write the WebSocket.Listener that will process the incoming messages. This is an interface that contains a bunch of methods with default implementations. The following code simply overrides onOpen(), onText(), and onClose(). Gluing the WebSocket listener and the preceding code will result in the following application:

public class Main {  
  
 public static void main(String[] args) throws   
 InterruptedException, ExecutionException, TimeoutException {  
  
 Listener wsListener = new Listener() {  
  
 @Override  
 public CompletionStage<?> onText(WebSocket webSocket,  
 CharSequence data, boolean last) {  
 System.out.println("Received data: " + data);  
  
 return Listener.super.onText(webSocket, data, last);  
 }  
  
 @Override  
 public void onOpen(WebSocket webSocket) {  
 System.out.println("Connection is open ...");  
 Listener.super.onOpen(webSocket);  
 }  
  
 @Override  
 public CompletionStage<? > onClose(WebSocket webSocket,  
 int statusCode, String reason) {  
 System.out.println("Closing connection: " +  
 statusCode + " " + reason);  
  
 return Listener.super.onClose(webSocket, statusCode, reason);  
 }  
 };  
  
 HttpClient client = HttpClient.newHttpClient();  
  
 WebSocket webSocket = client.newWebSocketBuilder()  
 .buildAsync(URI.create(  
 "ws://stream.meetup.com/2/rsvps"), wsListener)  
 .get(10, TimeUnit.SECONDS);  
  
 TimeUnit.SECONDS.sleep(10);  
  
 webSocket.sendClose(WebSocket.NORMAL\_CLOSURE, "ok");  
 }  
}Copy

This application will run for 10 seconds and will produce output similar to the following:

Connection is open ...  
  
Received data: {"visibility":"public","response":"yes","guests":0,"member":{"member\_id":267133566,"photo":"https:\/\/secure.meetupstatic.com\/photos\/member\/8\/7\/8\/a\/thumb\_282154698.jpeg","member\_name":"SANDRA MARTINEZ"},"rsvp\_id":1781366945...  
  
Received data: {"visibility":"public","response":"yes","guests":1,"member":{"member\_id":51797722,...  
...Copy

After 10 seconds, the application is disconnected from the WebSocket endpoint.

Summary

Our job is done! This was the last problem in this chapter. Now, we have reached the end of this book. It looks like the new HTTP Client and WebSocket APIs are pretty cool. They come with substantial flexibility and versatility, they are pretty intuitive, and they manage to successfully hide a lot of painful details that we don't want to deal with during development.

Download the applications from this chapter to view the results and additional details.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

[Imagen que contiene Sitio web

Descripción generada automáticamente](https://www.packtpub.com/in/application-development/learn-java-12-programming)

**Learn Java 12 Programming**  
Nick Samoylov

ISBN: 978-1-78995-705-1

* Learn and apply object-oriented principles
* Gain insights into data structures and understand how they are used in Java
* Explore multithreaded, asynchronous, functional, and reactive programming
* Add a user-friendly graphic interface to your application
* Find out what streams are and how they can help in data processing
* Discover the importance of microservices and use them to make your apps robust and scalable
* Explore Java design patterns and best practices to solve everyday problems
* Learn techniques and idioms for writing high-quality Java code

[Interfaz de usuario gráfica, Sitio web

Descripción generada automáticamente](https://www.packtpub.com/application-development/java-11-and-12-new-features)

**Java 11 and 12 - New Features**  
Mala Gupta

ISBN: 978-1-78913-327-1

* Study type interference and how to work with the var type
* Understand Class-Data Sharing, its benefits, and limitations
* Discover platform options to reduce your application’s launch time
* Improve application performance by switching garbage collectors
* Get up to date with the new Java release cadence
* Define and assess decision criteria for migrating to a new version of Java

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Insert15

Insert16

Insert16

Insert17

Insert18

Insert19

Insert20