*Parallel data processing and performance*

For instance, before Java 7, processing a collection of data in parallel was extremely cumbersome. In particular, we’ll demonstrate that the way a parallel stream gets divided into chunks, before processing the different chunks in parallel, can in some cases be the origin of these incorrect and apparently unexplainable results.

***Parallel streams***

A *parallel* stream is a stream that splits its elements into multiple chunks, processing each chunk

with a different thread.

public long sequentialSum(long n) {

return Stream.iterate(1L, i -> i + 1)

.limit(n)

.**reduce**(0L, Long::sum);

}

In more traditional Java terms, this code is equivalent to its iterative counterpart:

public long iterativeSum(long n) {

long result = 0;

for (long i = 1L; i <= n; i++) {

result += i;

}

return result;

}

This operation seems to be a good candidate to use **parallelization**, especially for large values of *n*. But where do you start? Do you synchronize on the result variable? How many threads do you use? Who does the generation of numbers? Who adds them up?

***Turning a sequential stream into a parallel one***

You can make the former functional reduction process (summing) run in parallel by

public long parallelSum(long n) {

return Stream.iterate(1L, i -> i + 1)

.limit(n)

.**parallel**() //**Turns the stream into a parallel one**

.**reduce**(0L, Long::sum);

}

The difference is that the stream is now **internally divided into multiple chunks**.

As a result, the reduction operation can work on the various chunks independently and in parallel.

Note that, in reality, calling the method parallel on a sequential stream doesn’t imply any concrete transformation on the stream itself. Internally, a boolean flag is set to signal that you want to run in parallel all the operations that follow the invocation to parallel. Similarly, you can turn a parallel stream into a sequential one by invoking the method sequential on it. Note that you might think that by combining these two methods you could achieve finer-grained control over which operations you want

to perform in parallel and which ones sequentially while traversing the stream. For example, you could do something like the following:

stream.parallel()

.filter(...)

.**sequential**()

.map(...)

.**parallel**()

.reduce();

But the last call to parallel or sequential wins and affects the pipeline globally. In this example, the pipeline will be executed in parallel because that’s the last call in the pipeline.

Configuring the thread pool used by parallel streams

Looking at the stream’s parallel method, you may wonder where the threads used

by the parallel stream come from, how many there are, and how you can customize

the process.

Parallel streams internally use the default ForkJoinPool (you’ll learn more about the fork/join framework in section 7.2), which by default has as many threads as you have processors, as returned by Runtime.getRuntime().available- Processors().

But you can change the size of this pool using the system property java.util

.concurrent.ForkJoinPool.common.parallelism, as in the following example:

System.setProperty("**java.util.concurrent.ForkJoinPool.common.parallelism**", "12");

This is a global setting, so it will affect all the parallel streams in your code. Conversely, it currently isn’t possible to specify this value for a single parallel stream. In general, having the size of the ForkJoinPool equal to the number of processors on your machine is a meaningful default, and we strongly suggest that you not modify it unless you have a good reason for doing so.

***Measuring stream performance***

We claimed that the parallelized summing method should perform better than the sequential and the iterative methods.

Nevertheless, in software engineering, guessing is never a good idea! When optimizing performance, you should always follow three golden rules: measure, measure, measure. To this purpose we will implement a microbenchmark using a library called Java Microbenchmark Harness (JMH).

This is a toolkit that helps to create, in a simple, annotation-based way, reliable microbenchmarks

for Java programs and for any other language targeting the Java Virtual Machine (JVM).

In fact, developing correct and meaningful benchmarks for programs running on the JVM is not an easy task, because there are many factors to consider like the warm-up time required by HotSpot to optimize the bytecode and the overhead introduced by the garbage collector. If you’re using Maven as your build tool, then to start using JMH in your project you add a couple of dependencies to your pom.xml file (which defines the Maven build process).

When you compile this class, the Maven plugin configured before generates a second JAR file named benchmarks.jar that you can run as follows:

java -jar ./target/benchmarks.jar ParallelStreamBenchmark

Running this second benchmark (possibly having commented out the first one to avoid running it again) on our testing machine, we obtained the following result:

Benchmark Mode Cnt Score Error Units

ParallelStreamBenchmark.**iterativeSum** avgt 40 3.278 ± 0.192 ms/op

This confirmed our expectations: the iterative version is almost 40 times faster than the one using the sequential stream for the reasons we anticipated. Now let’s do the same with the version using the parallel stream, also adding that method to our benchmarking class. We obtained the following outcome:

Benchmark Mode Cnt Score Error Units

ParallelStreamBenchmark.**parallelSum** avgt 40 604.059 ± 55.288 ms/op

This is quite disappointing: the parallel version of the summing method isn’t taking any advantage of our quad-core CPU and is around five times slower than the sequential one. How can you explain this unexpected result? Two issues are mixed together:

 iterate generates boxed objects, which have to be unboxed to numbers before

they can be added.

 iterate is difficult to divide into independent chunks to execute in parallel.

The second issue is particularly interesting because you need to keep a mental model that some stream operations are more parallelizable than others. Specifically, the iterate operation is hard to split into chunks that can be executed independently, because the input of one function application always depends on the result of the previous application.

When misused (for example, using an operation that’s not parallelfriendly, like iterate) it can worsen the overall performance of your programs, so it’s mandatory to understand what happens behind the scenes when you invoke that

apparently magic parallel method.

USING MORE SPECIALIZED METHODS

So how can you use your multicore processors and use the stream to perform a parallel sum in an effective way? We discussed a method called LongStream.rangeClosed in chapter 5. This method has two benefits compared to iterate:

 LongStream.rangeClosed works on primitive long numbers directly so there’s no boxing and unboxing overhead.

 LongStream.rangeClosed produces ranges of numbers, which can be easily split into independent chunks. For example, the range 1–20 can be split into 1–5, 6–10, 11–15, and 16–20.

Nevertheless, keep in mind that parallelization doesn’t come for free. The parallelization process itself requires you to recursively partition the stream, assign the reduction operation of each substream to a different thread, and then combine the results of these operations in a single value. But moving data between multiple cores is also more expensive than you might expect, so it’s important that work to be done in parallel on another core takes longer than the time required to transfer the data from one core to another. In general, there are many cases where it isn’t possible or convenient

to use parallelization. But before you use a parallel stream to make your code faster, you have to be sure that you’re using it correctly; it’s not helpful to produce a result in less time if the result will be wrong. Let’s look at a common pitfall.

***Using parallel streams effectively***

In general, it’s impossible (and pointless) to try to give any quantitative hint on when to use a parallel stream, because any specific criterion such as “only when the stream contains more than a thousand elements” could be correct for a specific operation running on a specific machine, but completely wrong in a marginally different context. Nonetheless,

it’s at least possible to provide some qualitative advice that could be useful when deciding whether it makes sense to use a parallel stream in a certain situation:

 If in doubt, measure. Turning a sequential stream into a parallel one is trivial but not always the right thing to do. As we already demonstrated in this section, a parallel stream isn’t always faster than the corresponding sequential version.

Moreover, parallel streams can sometimes work in a counterintuitive way, so the first and most important suggestion when choosing between sequential and parallel streams is to always check their performance with an appropriate

benchmark.

 Watch out for boxing. Automatic boxing and unboxing operations can dramatically hurt performance. Java 8 includes primitive streams (IntStream, LongStream, and DoubleStream) to avoid such operations, so use them when possible.

 Some operations naturally perform worse on a parallel stream than on a sequential stream. In particular, operations such as limit and findFirst that rely on the order of the elements are expensive in a parallel stream. For example,

findAny will perform better than findFirst because it isn’t constrained to operate in the encounter order. You can always turn an ordered stream into an unordered stream by invoking the method unordered on it. For instance, if you

need *N* elements of your stream and you’re not necessarily interested in the *first N* ones, calling limit on an unordered parallel stream may execute more efficiently than on a stream with an encounter order (for example, when the

source is a List).

 Consider the total computational cost of the pipeline of operations performed by the stream. With *N* being the number of elements to be processed and *Q* the approximate cost of processing one of these elements through the stream pipeline,

the product of *N\*Q* gives a rough qual itative estimation of this cost. A higher value for *Q* implies a better chance of good performance when using a parallel stream.

 For a small amount of data, choosing a parallel stream is almost never a winning decision. The advantages of processing in parallel only a few elements aren’t enough to compensate for the additional cost introduced by the parallelization process.

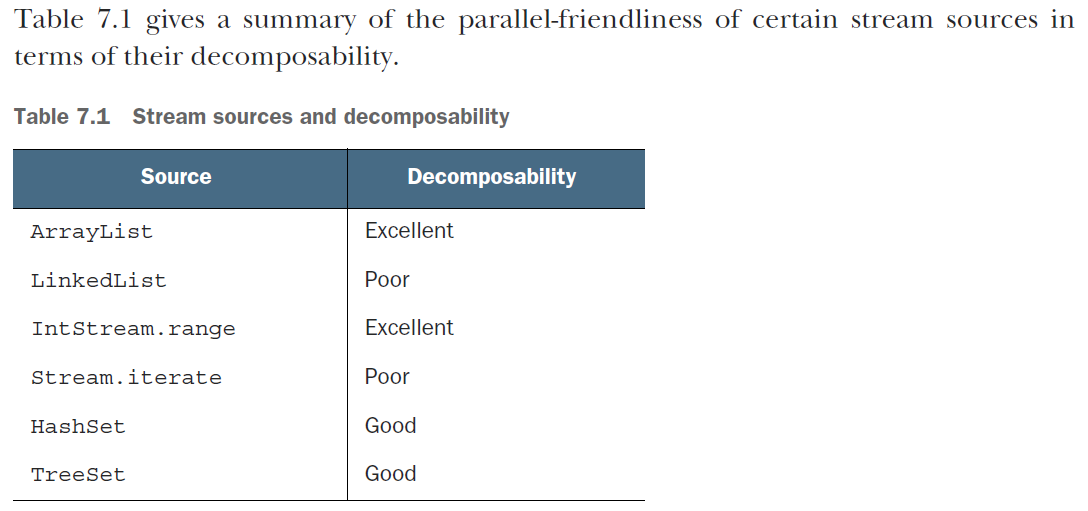
 Take into account how well the data structure underlying the stream decomposes. For instance, an ArrayList can be split much more efficiently than a LinkedList, because the first can be evenly divided without traversing it, as it’s

necessary to do with the second. Also, the primitive streams created with the range factory method can be decomposed quickly. Finally, as you’ll learn in section 7.3, you can get full control of this decomposition process by implementing your own Spliterator.

 The characteristics of a stream, and how the intermediate operations through the pipeline modify them, can change the performance of the decomposition process. For example, a SIZED stream can be divided into two equal parts, and

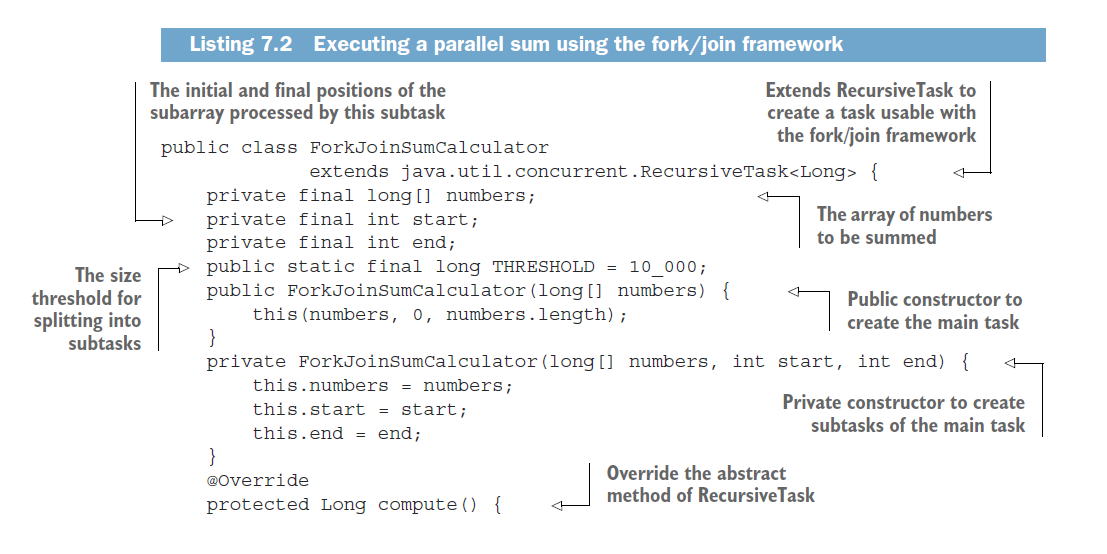
then each part can be processed in parallel more effectively, but a filter operation can throw away an unpredictable number of elements, making the size of the stream itself unknown.

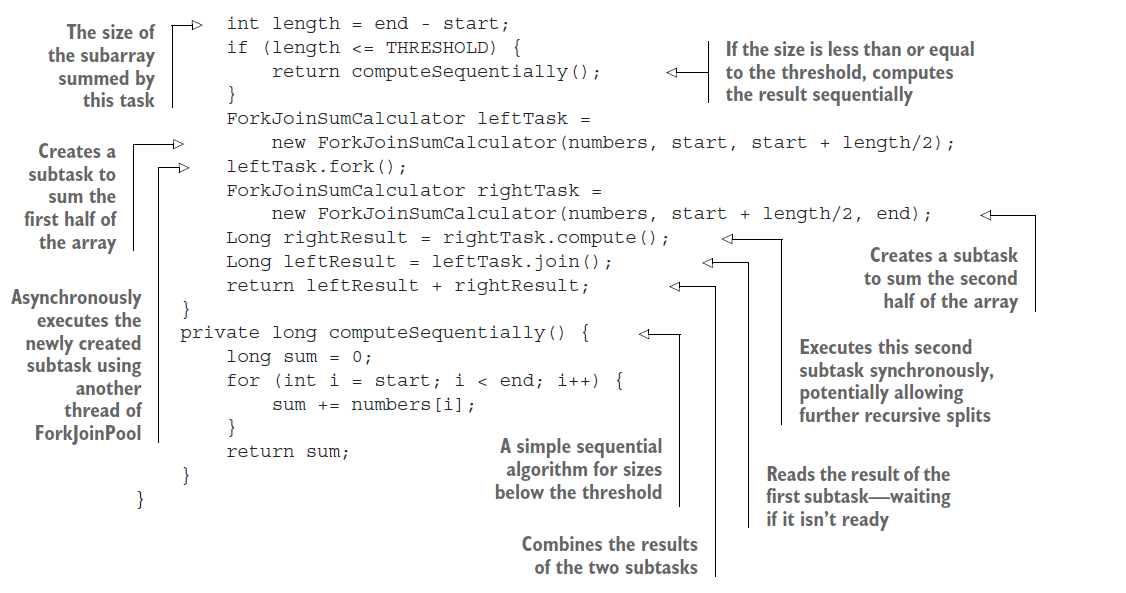
 Consider whether a terminal operation has a cheap or expensive merge step (for example, the combiner method in a Collector). If this is expensive, then the cost caused by the combination of the partial results generated by each substream can outweigh the performance benefits of a parallel stream.



***The fork/join framework***

The fork/join framework was designed to recursively split a parallelizable task into smaller tasks and then combine the results of each subtask to produce the overall result. It’s an implementation of the ExecutorService interface, which distributes those subtasks to worker threads in a thread pool, called ForkJoinPool.





***Best practices for using the fork/join framework***

Even though the fork/join framework is relatively easy to use, unfortunately it’s also easy to misuse. Here are a few best practices to use it effectively:

 Invoking the join method on a task blocks the caller until the result produced by that task is ready. For this reason, it’s necessary to call it after the computation of both subtasks has been started. Otherwise, you’ll end up with a slower

and more complex version of your original sequential algorithm because every subtask will have to wait for the other one to complete before starting.

 The invoke method of a ForkJoinPool shouldn’t be used from within a RecursiveTask. Instead, you should always call the methods compute or fork directly; only sequential code should use invoke to begin parallel computation.

 Calling the fork method on a subtask is the way to schedule it on the ForkJoinPool. It might seem natural to invoke it on both the left and right subtasks, but this is less efficient than directly calling compute on one of them.

Doing this allows you to reuse the same thread for one of the two subtasks and avoid the overhead caused by the unnecessary allocation of a further task on the pool.

 Debugging a parallel computation using the fork/join framework can be tricky. In particular, it’s ordinarily quite common to browse a stack trace in your favorite IDE to discover the cause of a problem, but this can’t work with a fork/join computation because the call to compute occurs in a different thread than the conceptual caller, which is the code that called fork.

 As you’ve discovered with parallel streams, you should never take for granted that a computation using the fork/join framework on a multicore processor is faster than the sequential counterpart. We already said that a task should be

decomposable into several independent subtasks in order to be parallelizable with a relevant performance gain. All of these subtasks should take longer to execute than forking a new task; one idiom is to put I/O into one subtask and

computation into another, thereby overlapping computation with I/O. Moreover, you should consider other things when comparing the performance of the sequential and parallel versions of the same algorithm. Like any other

Java code, the fork/join framework needs to be “warmed up,” or executed, a few times before being optimized by the JIT compiler. This is why it’s always important to run the program multiple times before to measure its performance,

as we did in our harness. Also be aware that optimizations built into the compiler could unfairly give an advantage to the sequential version (for example, by performing dead code analysis—removing a computation that’s

never used). The fork/join splitting strategy deserves one last note: you must choose the criteria used to decide if a given subtask should be further split or is small enough to be evaluated sequentially. We’ll give some hints about this in the next section.

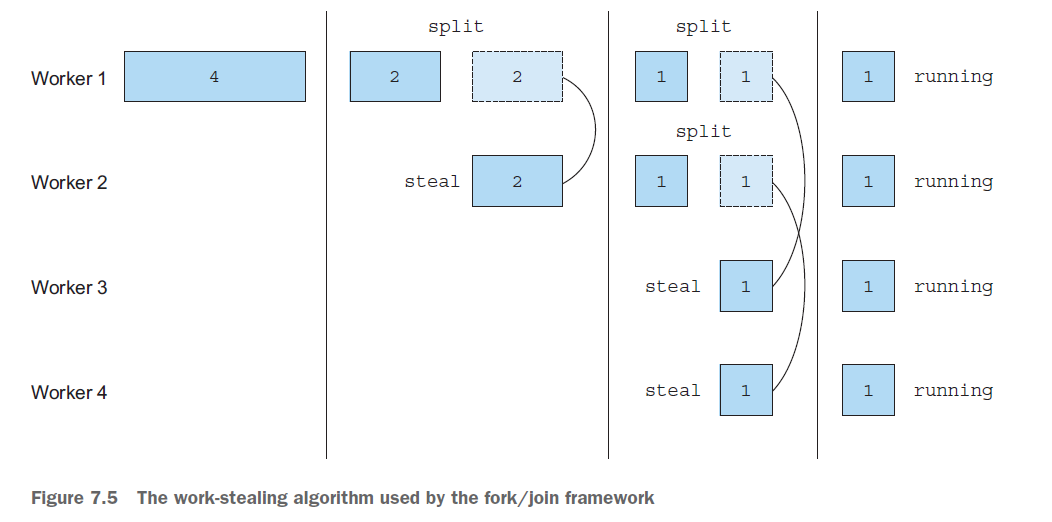
***Work stealing***

But forking a quite large number of fine-grained tasks is in general a winning choice. This is because ideally you want to partition the workload of a parallelized task in such a way that each subtask takes exactly the same amount of time, keeping all the cores of your CPU equally busy. Unfortunately, especially in cases closer to real-world

scenarios than the straightforward example we presented here, the time taken by each subtask can dramatically vary either due to the use of an inefficient partition strategy or because of unpredictable causes like slow access to the disk or the need to coordinate the execution with external services. The fork/join framework works around this problem with a technique called *work stealing*. In practice, this means that the tasks are more or less evenly divided on all the

threads in the ForkJoinPool. Each of these threads holds a doubly linked queue of the tasks assigned to it, and as soon as it completes a task it pulls another one from the head of the queue and starts executing it.

More generally, this work-stealing algorithm is used to redistribute and balance the tasks among the worker threads in the pool.



***Spliterator***

The Spliterator is another new interface added to Java 8; its name stands for “splitable iterator.” Like Iterators, Spliterators are used to traverse the elements of a source, but they’re also designed to do this in parallel.

Java 8 already provides a default Spliterator implementation for all the data structures included in its Collections

Framework. The Collection interface now provides a default method spliterator()

***Summary***

 Internal iteration allows you to process a stream in parallel without the need to explicitly use and coordinate different threads in your code.

 Even if processing a stream in parallel is so easy, there’s no guarantee that doing so will make your programs run faster under all circumstances. Behavior and performance of parallel software can sometimes be counterintuitive, and for this reason it’s always necessary to measure them and be sure that you’re not slowing your programs down.

 Parallel execution of an operation on a set of data, as done by a parallel stream, can provide a performance boost, especially when the number of elements to be processed is huge or the processing of each single element is particularly

time consuming.

 From a performance point of view, using the right data structure, for instance, employing primitive streams instead of nonspecialized ones whenever possible, is almost always more important than trying to parallelize some operations.

 The fork/join framework lets you recursively split a parallelizable task into smaller tasks, execute them on different threads, and then combine the results of each subtask in order to produce the overall result.

 Spliterators define how a parallel stream can split the data it traverses.

*Effective programming with streams and lambdas*

***Collection factories***

***How would you create a small list of elements in Java? You might want to group the***

names of your friends who are going on a holiday, for example. Here’s one way:

List<String> friends = new ArrayList<>();

friends.add("Raphael");

friends.add("Olivia");

friends.add("Thibaut");

But that’s quite a few lines to write for storing three strings! A more convenient way to

write this code is to use the Arrays.asList() factory method:

List<String> friends

= Arrays.asList("Raphael", "Olivia", "Thibaut");

You get a fixed-sized list that you can update, but not add elements to or remove elements

from. Attempting to add elements, for example, results in an Unsupported-

ModificationException, but updating by using the method set is allowed:

List<String> friends = Arrays.asList("Raphael", "Olivia");

friends.set(0, "Richard");

friends.add("Thibaut");

This behavior seems slightly surprising because the underlying list is backed by a

mutable array of fixed size.

How about a Set? Unfortunately, there’s no Arrays.asSet() factory method, so

you need another trick. You can use the HashSet constructor, which accepts a List:

Set<String> friends "

= new HashSet<>(Arrays.asList("Raphael", "Olivia", Thibaut"));

Alternatively you could use the Streams API:

Set<String> friends

= Stream.of("Raphael", "Olivia", "Thibaut")

.collect(Collectors.toSet());

Both solutions, however, are far from elegant and involve unnecessary object allocations

behind the scenes. Also note that you get a mutable Set as a result.

***List factory***

You can create a list simply by calling the factory method List.of:

List<String> friends = List.of("Raphael", "Olivia", "Thibaut");

System.out.println(friends);

friends.add("Chih-Chun");//try adding new element

Running this code results in a java.lang.**UnsupportedOperationException**. In fact,

the list that’s produced is immutable. Replacing an item with the set() method throws

a similar exception. This restriction is a good thing, however, as it protects you from unwanted mutations

of the collections.

Nothing is stopping you from having elements that are mutable themselves. If you need a mutable list, you can still instantiate one manually. Finally, note that to prevent unexpected bugs and enable a more-compact internal representation, null elements are disallowed. After all, you saw in previous chapters that

you can use the Collectors.toList() collector to transform a stream into a list.

***Set factory***

As with List.of, you can create an immutable Set out of a list of elements:

Set<String> friends = Set.of("Raphael", "Olivia", "Thibaut", "Olivia");

System.out.println(friends);

If you try to create a Set by providing a duplicated element, you receive an **Illegal**

**ArgumentException**.

***Map factories***

Creating a map is a bit more complicated than creating lists and sets because you have

to include both the key and the value. And must be even number.

Map<String, Integer> ageOfFriends

= Map.of("Raphael", 30, "Olivia", 25, "Thibaut", 26);

System.out.println(ageOfFriends);

This method is convenient if you want to create a small map of up to ten keys and values.

To go beyond this, use the alternative factory method called Map.ofEntries,

which takes Map.Entry<K, V> objects but is implemented with varargs. This method

requires additional object allocations to wrap up a key and a value:

import static java.util.Map.entry;

Map<String, Integer> ageOfFriends

= Map.ofEntries(entry("Raphael", 30),

entry("Olivia", 25),

entry("Thibaut", 26));

System.out.println(ageOfFriends);

***Working with List and Set***

Java 8 introduced a couple of methods into the List and Set interfaces:

 removeIf removes element matching a predicate. It’s available on all classes

that implement List or Set (and is inherited from the Collection interface).

 replaceAll is available on List and replaces elements using a (UnaryOperator)

function.

 sort is also available on the List interface and sorts the list itself.

All these methods mutate the collections on which they’re invoked. In other words, they change the collection itself, unlike stream operations, which produce a new (copied) result.

***removeIf***

Consider the following code, which tries to remove transactions that have a reference

code starting with a digit:

for (Transaction transaction : transactions) {

if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {

transactions.remove(transaction);

}

}

Can you see the problem? Unfortunately, this code may result in a **ConcurrentModificationException**. Why? Under the hood, the for-each loop uses an Iterator object, so the code executed is as follows:

for (Iterator<Transaction> iterator = transactions.iterator();

iterator.hasNext(); ) {

Transaction transaction = iterator.next();

if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {

transactions.remove(transaction);

}

}

Notice that two separate objects manage the collection:

 The Iterator object, which is querying the source by using next() and hasNext()

 The Collection object itself, which is removing the element by calling remove()

As a result, the state of the iterator is no longer synced with the state of the collection, and vice versa. To solve this problem, you have to use the Iterator object explicitly and call its remove() method:

for (Iterator<Transaction> iterator = transactions.iterator();

iterator.hasNext(); ) {

Transaction transaction = iterator.next();

if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {

**iterator.remove();**

}

}

This code has become fairly verbose to write. This code pattern is now directly expressible with the Java 8 removeIf method, which is not only simpler but also protects you from these bugs. It takes a predicate indicating which elements to remove:

transactions.**removeIf**(transaction → Character.isDigit(transaction.getReferenceCode().charAt(0)));

Sometimes, though, instead of removing an element, you want to replace it. For this purpose, Java 8 added replaceAll.

***replaceAll***

The replaceAll method on the List interface lets you replace each element in a list with a new one. Using the Streams API, you could solve this problem as follows:

referenceCodes.stream()

.map(code -> Character.toUpperCase(code.charAt(0)) + code.substring(1))

.collect(Collectors.toList()).forEach(System.out::println);

This code results in a new collection of strings, however. You want a way to update the existing collection. You can use a ListIterator object as follows (supporting a set() method to replace an element):

for (ListIterator<String> iterator = referenceCodes.listIterator();

iterator.hasNext(); ) {

String code = iterator.next();

iterator.set(Character.toUpperCase(code.charAt(0)) + code.substring(1));

}

As you can see, this code is fairly verbose. In addition, as we explained earlier, using Iterator objects in conjunction with collection objects can be error-prone by mixing iteration and modification of the collection. In Java 8, you can simply write

referenceCodes.replaceAll(code -> Character.toUpperCase(code.charAt(0)) +

code.substring(1));

***forEach***

Iterating over the keys and values of a Map has traditionally been awkward. In fact, you needed to use an iterator of a Map.Entry<K, V> over the entry set of a Map:

for(Map.Entry<String, Integer> entry: ageOfFriends.entrySet()) {

String friend = entry.getKey();

Integer age = entry.getValue();

System.out.println(friend + " is " + age + " years old");

}

Since Java 8, the Map interface has supported the forEach method, which accepts a BiConsumer, taking the key and value as arguments. Using forEach makes your code more concise:

ageOfFriends.forEach((friend, age) -> System.out.println(friend + " is " +

age + " years old"));

***Sorting***

Two new utilities let you sort the entries of a map by values or keys:

 Entry.comparingByValue

 Entry.comparingByKey

HashMap and Performance

The internal structure of a HashMap was updated in Java 8 to improve performance. Entries of a map typically are stored in buckets accessed by the generated hashcode of the key. But if many keys return the same hashcode, performance deteriorates because buckets are implemented as LinkedLists with O(n) retrieval. Nowadays,

when the buckets become too big, they’re replaced dynamically with **sorted trees, which have O(log(n))** retrieval and improve the lookup of colliding elements. Note that this use of sorted trees is possible only when the keys are Comparable (such as String or Number classes).

Another common pattern is how to act when the key you’re looking up in the Map isn’t present. The new getOrDefault method can help.

***8.3.3 getOrDefault***

When the key you’re looking up isn’t present, you receive a null reference that you have to check against to prevent a NullPointerException. A common design style is to provide a default value instead. Now you can encode this idea more simply by using the **getOrDefault** method.

This method takes the key as the first argument and a default value (to be used when the key is absent from the Map) as the second argument:

***Compute patterns***

Sometimes, you want to perform an operation conditionally and store its result,

depending on whether a key is present or absent in a Map. You may want to cache the

result of an expensive operation given a key, for example. If the key is present, there’s

no need to recalculate the result. Three new operations can help:

 computeIfAbsent—If there’s no specified value for the given key (it’s absent or

its value is null), calculate a new value by using the key and add it to the Map.

 computeIfPresent—If the specified key is present, calculate a new value for it

and add it to the Map.

 compute—This operation calculates a new value for a given key and stores it in

the Map.

***Remove patterns***

You already know about the remove method that lets you remove a Map entry for a given key. Since Java 8, an overloaded version removes an entry only if the key is associated with a specific value.

***Replacement patterns***

Map has two new methods that let you replace the entries inside a Map:

 replaceAll—Replaces each entry’s value with the result of applying a

BiFunction. This method works similarly to replaceAll on a List, which you

saw earlier.

 Replace—Lets you replace a value in the Map if a key is present. An additional

overload replaces the value only if it the key is mapped to a certain value.

You could format all the values in a Map as follows:

Map<String, String> favouriteMovies = new HashMap<>();

favouriteMovies.put("Raphael", "Star Wars");

favouriteMovies.put("Olivia", "james bond");

favouriteMovies.replaceAll((friend, movie) -> movie.toUpperCase());

***Merge***

Suppose that you’d like to merge two intermediate Maps, perhaps two separate Maps for

two groups of contacts. You can use putAll as follows:

Map<String, String> **family** = Map.ofEntries(

entry("Teo", "Star Wars"), entry("Cristina", "James Bond"));

Map<String, String> **friends** = Map.ofEntries(entry("Raphael", "Star Wars"));

Map<String, String> everyone = new HashMap<>(family); //create new map with famility content

everyone.**putAll**(friends);//**Copies all the entries from friends into everyone**

System.out.println(everyone);

This code works as expected as long as you don’t have duplicate keys. If you require more flexibility in how values are combined, you can use the new merge method. This method takes a BiFunction to merge values that have a duplicate key.

You could use the **merge** method in combination with forEach to provide a way to deal with the conflict. The following code concatenates the string names of the two movies:

Map<String, String> everyone = new HashMap<>(**family**);

**friends**.forEach((k, v) → everyone.**merge**(k, v, (movie1, movie2) -> movie1 + " & " + movie2));

System.out.println(everyone);

Note that the merge method has a fairly complex way to deal with nulls, as specified in the Javadoc:

*If the specified key is not already associated with a value or is associated with null, [*merge*] associates it with the given non-null value. Otherwise, [*merge*] replaces the associated value with the [result] of the given remapping function, or removes [it] if the result is null.*

*You can also use merge to implement initialization checks. Suppose that you have a* Map for recording how many times a movie is watched. You need to check that the key representing the movie is in the map before you can increment its value:

Map<String, Long> moviesToCount = new HashMap<>();

String movieName = "JamesBond";

long count = moviesToCount.get(movieName);

if(count == null) {

moviesToCount.put(movieName, 1);

}

else {

moviesToCount.put(moviename, count + 1);

}

This code can be rewritten as

moviesToCount.merge(movieName, 1L, (key, count) -> count + 1L);

Iterator<Map.Entry<String, Integer>> iterator =

movies.entrySet().iterator();

while(iterator.hasNext()) {

Map.Entry<String, Integer> entry = iterator.next();

if(entry.getValue() < 10) {

iterator.remove();

}

}

System.out.println(movies);

You can use the removeIf method on the map’s entry set, which takes a predicate and removes the elements:

movies.entrySet().**removeIf**(entry -> entry.getValue() < 10);

***Improved ConcurrentHashMap***

The **ConcurrentHashMap** class was introduced to provide a more modern HashMap, which is also concurrency friendly. ConcurrentHashMap allows concurrent add and update operations that lock only certain parts of the internal data structure. Thus, read and write operations have improved performance compared with the synchronized

**Hashtable** alternative. (Note that the standard **HashMap** is unsynchronized.)

***Reduce and Search***

ConcurrentHashMap supports three new kinds of operations, reminiscent of what you

saw with streams:

 forEach—Performs a given action for each (key, value)

 reduce—Combines all (key, value) given a reduction function into a result

 search—Applies a function on each (key, value) until the function produc es a non-null result

ConcurrentHashMap<String, Long> map = new ConcurrentHashMap<>();

long parallelismThreshold = 1;

Optional<Integer> maxValue =

Optional.ofNullable(map.reduceValues(parallelismThreshold, Long::max));

***Counting***

The ConcurrentHashMap class provides a new method called mappingCount, which returns the number of mappings in the map as a long.

***Set views***

The ConcurrentHashMap class provides a new method called keySet that returns a view of the ConcurrentHashMap as a Set.

***Summary***

 Java 9 supports collection factories, which let you create small immutable lists, sets, and maps by using List.of, Set.of, Map.of, and Map.ofEntries.

 The objects returned by these collection factories are immutable, which means that you can’t change their state after creation.

 The List interface supports the default methods removeIf, replaceAll, and sort.

 The Set interface supports the default method removeIf.

 The Map interface includes several new default methods for common patterns and reduces the scope for bugs.

 ConcurrentHashMap supports the new default methods inherited from Map but provides thread-safe implementations for them.

*Refactoring, testing, and debugging*

***Refactoring for improved readability and flexibility***

From the start of this book, we’ve argued that lambda expressions let you write more concise and flexible code.

The code is more concise because lambda expressions let you represent a piece of behavior in a more compact form compared with using anonymous classes.

Your code is more flexible because lambda expressions encourage the style of behavior parameterization that we introduced in chapter 2. Your code can use and execute multiple behaviors passed as arguments to cope with requirement changes.

***Improving code readability***

What does it mean to improve the readability of code? Defining good readability can be subjective. The general view is that the term means “how easily this code can be understood by another human.” Improving code readability ensures that your code is understandable and maintainable by people other than you.

***From anonymous classes to lambda expressions***

The first simple refactoring you should consider is converting uses of anonymous classes implementing one single abstract method to lambda expressions.

Runnable r1 = new Runnable() {

public void run(){

System.out.println("Hello");

}

};

//refactor to

Runnable r2 = () -> System.out.println("Hello");

But converting anonymous classes to lambda expressions can be a difficult process in certain situations. First, the meanings of **this** and **super** are different for anonymous classes and lambda expressions. Inside an anonymous class, this refers to the anonymous class itself, but inside a lambda, it refers to the enclosing class. Second, anonymous classes are allowed to shadow variables from the enclosing class. Lambda expressions can’t (they’ll cause a compile error), as shown in the following code:

int a = 10;

Runnable r1 = new Runnable(){

public void run(){

int a = 2; //all OK

System.out.println(a);

}

};

Runnable r2 = () -> {

int a = 2; //compile error

System.out.println(a);

};

Suppose that you’ve declared a functional interface with the same signature as Runnable, here called Task (as might

occur when you need more-meaningful interface names in your domain model):

interface Task {

public void execute();

}

public static void doSomething(Runnable r){ r.run(); }

public static void doSomething(Task a){ r.execute(); }

Now you can pass an anonymous class implementing Task without a problem:

doSomething(new Task() {

public void execute() {

System.out.println("Danger danger!!");

}

});

But converting this anonymous class to a lambda expression results in an ambiguous method call, because both Runnable and Task are valid target types:

doSomething(() -> System.out.println("Danger danger!!"));//**Problem; both doSomething(Runnable) and doSomething(Task) match.**

**You can solve the ambiguity by providing an explicit cast (Task):**

doSomething((Task)() -> System.out.println("Danger danger!!"));

**Most integrated development** environments (IDEs)—such as NetBeans, Eclipse, and IntelliJ—support this

refactoring and automatically ensure that these gotchas don’t arise.

***From lambda expressions to method references***

Lambda expressions are great for short code that needs to be passed around.

***From imperative data processing to Streams***

Ideally, you should try to convert all code that processes a collection with typical data processing patterns with an iterator to use the Streams API instead. Why? The Streams API expresses more clearly the intent of a data processing pipeline. In addition, streams can be optimized behind the scenes, making use of short-circuiting and laziness as well as leveraging your multicore architecture, as we explained in chapter 7.

List<String> dishNames = new ArrayList<>();

for(Dish dish: menu) {

if(dish.getCalories() > 300){

dishNames.add(dish.getName());

}

}

The alternative, which uses the Streams API, reads more like the problem statement, and it can be easily parallelized:

menu.parallelStream()

.filter(d -> d.getCalories() > 300)

.map(Dish::getName)

.collect(toList());

Unfortunately, converting imperative code to the Streams API can be a difficult task, because you need to think about control-flow statements such as break, continue, and return and then infer the right stream operations to use. The good news is that some tools can help you with this task as well. The good news is that some tools (e.g., Lambda-

Ficator, https://ieeexplore.ieee.org/document/6606699) can help you with this task as well.

http://www.youtube.com/watch?v=EIyAflgHVpU

***Improving code flexibility***

We argued in chapters 2 and 3 that lambda expressions encourage the style of behavior parameterization.

This style lets you cope with requirement changes (creating multiple ways of filtering with a Predicate or comparing with a Comparator, for example).

***Refactoring object-oriented design patterns with lambdas***

New language features often make existing code patterns or idioms less popular. The introduction of the for-each loop in Java 5, for example, has replaced many uses of explicit iterators because it’s less error-prone and more concise.

A specific class of patterns is called design patterns.2 *Design patterns* are reusable blueprints, if you will, for common problems in designing software. They are rather like how construction engineers have a set of reusable solutions to construct bridges for specific scenarios (suspension bridge, arch bridge, and so on).

***Testing lambdas***

You’ve sprinkled your code with lambda expressions, and it looks nice and concise. But in most developer jobs, you’re paid not for writing nice code, but for writing code that’s correct.

***Testing the behavior of a visible lambda***

This code works nicely because the moveRightBy method is public and, therefore, can be tested inside the test case. But lambdas don’t have names (they’re anonymous functions, after all), and testing them in your code is tricky because you can’t refer to them by name.

Sometimes, you have access to a lambda via a field so that you can reuse it, and you’d like to test the logic encapsulated in that lambda. What can you do? You could test the lambda as you do when calling methods.

public class Point {

public final static Comparator<Point> compareByXAndThenY = comparing(Point::getX).thenComparing(Point::getY);

…

}

Here, you can call the compare

method on the Comparator object compareByXAndThenY with different arguments to

test whether its behavior is as intended:

@Test

public void testComparingTwoPoints() throws Exception {

Point p1 = new Point(10, 15);

Point p2 = new Point(10, 20);

int result = Point.compareByXAndThenY.compare(p1 , p2);

assertTrue(result < 0);

}

***Debugging***

A developer’s arsenal has two main old-school weapons for debugging problematic code:

 Examining the stack trace

 Logging

When your program fails, you get a ***stack trace***, which is a summary of how your program got to that failure, stack frame by stack frame. In other words, you get a valuable list of method calls up to when the failure appeared. This list helps you understand how the problem occurred. e.g.

The divideByZero method is reported correctly in the stack trace:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Debugging.divideByZero(Debugging.java:10)

at Debugging$$Lambda$1/999966131.apply(Unknown Source)

at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline

.java:193)

***Logging information***

Suppose that you’re trying to debug a pipeline of operations in a stream.

***Summary***

 Lambda expressions can make your code more readable and flexible.

 Consider converting anonymous classes to lambda expressions, but be wary of

subtle semantic differences such as the meaning of the keyword this and shadowing

of variables.

 Method references can make your code more readable compared with lambda

expressions.

 Consider converting iterative collection processing to use the Streams API.

 Lambda expressions can remove boilerplate code associated with several objectoriented

design patterns, such as strategy, template method, observer, chain of

responsibility, and factory.

 Lambda expressions can be unit-tested, but in general, you should focus on testing

the behavior of the methods in which the lambda expressions appear.

 Consider extracting complex lambda expressions into regular methods.

 Lambda expressions can make stack traces less readable.

 The peek method of a stream is useful for logging intermediate values as they

flow past certain points of a stream pipeline.

*Domain-specific languages using lambdas*

***Domain experts*** *can be involved in the software* development process and verify the correctness of the software from a business point of view. As a result, bugs and misunderstandings can be caught early on.

To achieve this result, it’s common to express the application’s business logic through a domain-specific language (DSL). A *DSL* is a small, usually non-generalpurpose programming language explicitly tailored for a specific domain.

You may be familiar with Maven and Ant, for example. You can see them as DSLs for expressing build processes.

You can see them as DSLs for expressing build processes. You’re also familiar with HTML, which is a language tailored to define the structure of a web page.

Historically, due to its rigidity and excessive verbosity, Java has never been popular for implementing a compact DSL that’s also suitable to be read by non-technical people.

***A specific language for your domain***

A DSL is a custom-built language designed to solve a problem for a specific business domain.

A DSL isn’t a general-purpose programming language; it restricts the operations and vocabulary available to a specific domain, which means that you have less to worry about and can invest more attention in solving the business problem at hand.

***Pros and cons of DSLs***

DSLs, like other technologies and solutions in software development, aren’t silver bullets.

DSLs offer the following **benefits**:

 *Conciseness*—An API that conveniently encapsulates the business logic allows you to avoid repetition, resulting in code that’s less verbose.

 *Readability*—Using words that belong to the vocabulary of the domain makes the code understandable even by domain non-experts. Consequently, code and domain knowledge can be shared across a wider range of members of the organization.

 *Maintainability*—Code written against a well-designed DSL is easier to maintain and modify. Maintainability is especially important for business-related code, which is the part of the application that may change most frequently.

 *Higher level of abstraction*—The operations available in a DSL work at the same level of abstraction as the domain, thus hiding the details that aren’t strictly related to the domain’s problems.

 *Focus*—Having a language designed for the sole purpose of expressing the rules of the business domain helps programmers stay focused on that specific part of the code. The result is increased productivity.

 *Separation of concerns*—Expressing the business logic in a dedicated language makes it easier to keep the business-related code isolated from the infrastructural part of the application. The result is code that’s easier to maintain.

Conversely, introducing a DSL into your code base can have a few **disadvantages**:

 *Difficulty of DSL design*—It’s hard to capture domain knowledge in a concise limited language.

 *Development cost*—Adding a DSL to your code base is a long-term investment with a high up-front cost, which could delay your project in its early stages. In addition, the maintenance of the DSL and its evolution add further engineering overhead.

 *Additional indirection layer*—A DSL wraps your domain model in an additional layer that has to be as thin as possible to avoid incurring performance problems.

 *Another language to learn*—Nowadays, developers are used to employing multiple languages. Adding a DSL to your project, however, implicitly implies that you and your team have one more language to learn. Worse, if you decide to

have multiple DSLs covering different areas of your business domain, combining them in a seamless way could be hard, because DSLs tend to evolve independently.

 *Hosting-language limitations*—Some general-purpose programming languages (Java is one of them) are known for being verbose and having rigid syntax. These languages make it difficult to design a user-friendly DSL. In fact, DSLs

developed on top of a verbose programming language are constrained by the cumbersome syntax and may not be nice to read. The introduction of lambda expression in Java 8 offers a powerful new tool to mitigate this problem.

***Different DSL solutions available on the JVM***

***The most common way to categorize DSLs, introduced by Martin Fowler, is to distinguish***

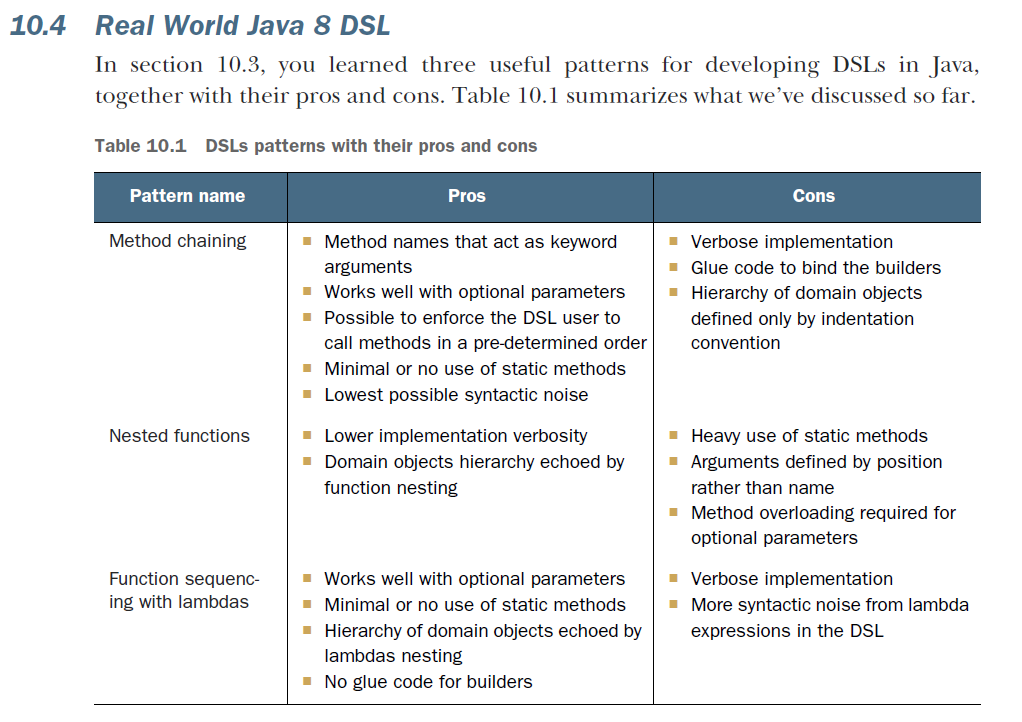
between **internal** and **external** DSLs. Internal DSLs (also known as embedded DSLs) are implemented on top of the existing hosting language (which could be plain Java code), whereas external DSLs are called stand-alone because they’re developed from scratch with a syntax that’s independent of the hosting language.

Moreover, the JVM gives you a third possibility that falls between an internal and an external DSL: another general-purpose programming language that also runs on the JVM but is more flexible and expressive than Java, such as Scala or Groovy. We refer to this third alternative as a **polyglot DSL**.

***Patterns and techniques to create DSLs in Java***

A DSL provides a friendly, readable API to work with a particular domain model.

|  |  |
| --- | --- |
| ***Using nested functions***  The *nested function* DSL pattern takes its name from the fact that it populates the  domain model by using functions that are nested within other functions. The following  listing illustrates the DSL style resulting from this approach  Order order = order("BigBank",  buy(80,  stock("IBM", on("NYSE")),  at(125.00)),  sell(50,  stock("GOOGLE", on("NASDAQ")),  at(375.00))  ); | ***Method chaining***  The first style of DSL to explore is one of the most common. It allows you to define a trading order with a single chain of method invocations. The following listing shows an example of this type of DSL.  Order order = forCustomer( "BigBank" )  .buy( 80 )  .stock( "IBM" )  .on( "NYSE" )  .at( 125.00 )  .sell( 50 )  .stock( "GOOGLE" )  .on( "NASDAQ" )  .at( 375.00 )  .end(); |



***jOOQ***

SQL is one of the most common and widely used DSLs. For this reason, it shouldn’t be surprising that there’s a Java library providing a nice DSL to write and execute SQL queries. jOOQ is an internal DSL that implements SQL as a type-safe embedded language directly in Java.

SELECT \* FROM BOOK

WHERE BOOK.PUBLISHED\_IN = 2016

ORDER BY BOOK.TITLE

can be written using the jOOQ DSL like this:

create.selectFrom(BOOK)

.where(BOOK.PUBLISHED\_IN.eq(2016))

.orderBy(BOOK.TITLE)

Another nice feature of the jOOQ DSL is the possibility of using it in combination with the Stream API.

**Starts manipulating data fetched from database with Stream API**

**Class.forName("org.h2.Driver");**

try (Connection c =

getConnection("jdbc:h2:~/sql-goodies-with-mapping", "sa", "")) {

DSL.using(c)

.select(BOOK.AUTHOR, BOOK.TITLE)

.where(BOOK.PUBLISHED\_IN.eq(2016))

.orderBy(BOOK.TITLE)

.fetch()

.stream()

.collect(groupingBy(

r -> r.getValue(BOOK.AUTHOR),

LinkedHashMap::new,

mapping(r -> r.getValue(BOOK.TITLE), toList())))

.forEach((author, titles) ->

System.out.println(author + " is author of " + titles));

}

***Cucumber***

Behavior-driven development (BDD) is an extension of test-driven development that uses a simple domain-specific scripting language made of structured statements that describe various business scenarios.

Cucumber, like other BDD frameworks, translates these statements into executable test cases.

BDD also focuses the development effort on the delivery of prioritized, verifiable business value and bridges the gap

between domain experts and programmers by making them share a business vocabulary.

Use Cucumber’s scripting language as follows to define a simple business scenario:

Feature: Buy stock

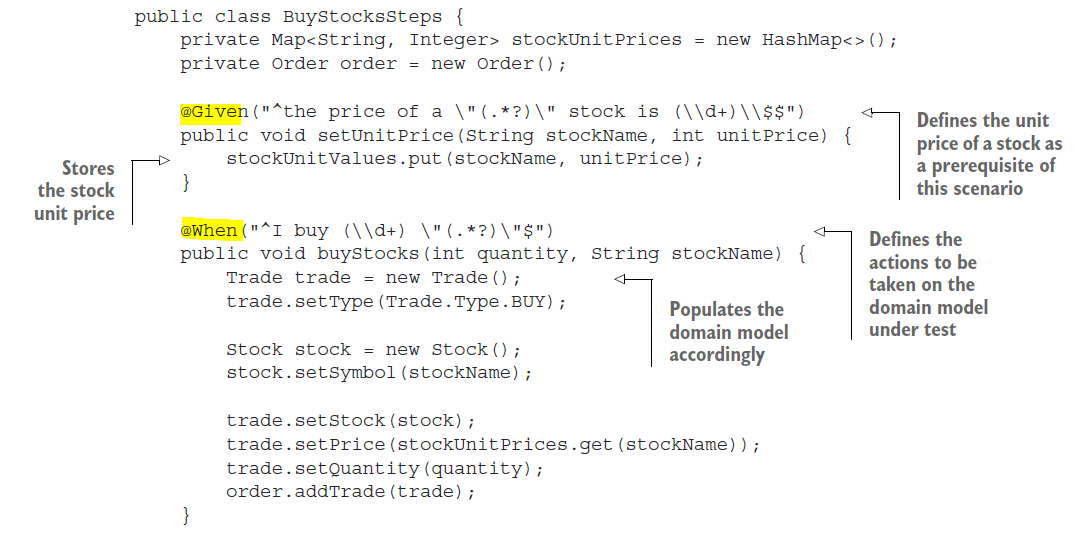
Scenario: Buy 10 IBM stocks

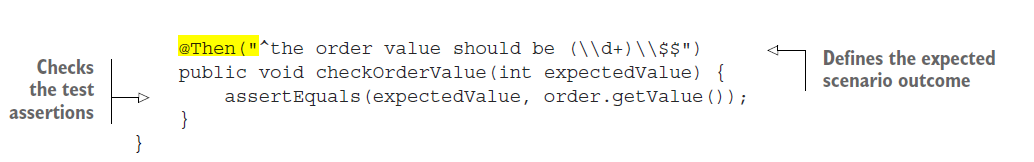
Given the price of a "IBM" stock is 125$

When I buy 10 "IBM"

Then the order value should be 1250$

Cucumber uses notation that’s divided into three parts: the definition of prerequisites (**Given**), the actual calls to the domain objects under test, and (**When**) the assertions checking the outcome of the test case (**Then**).



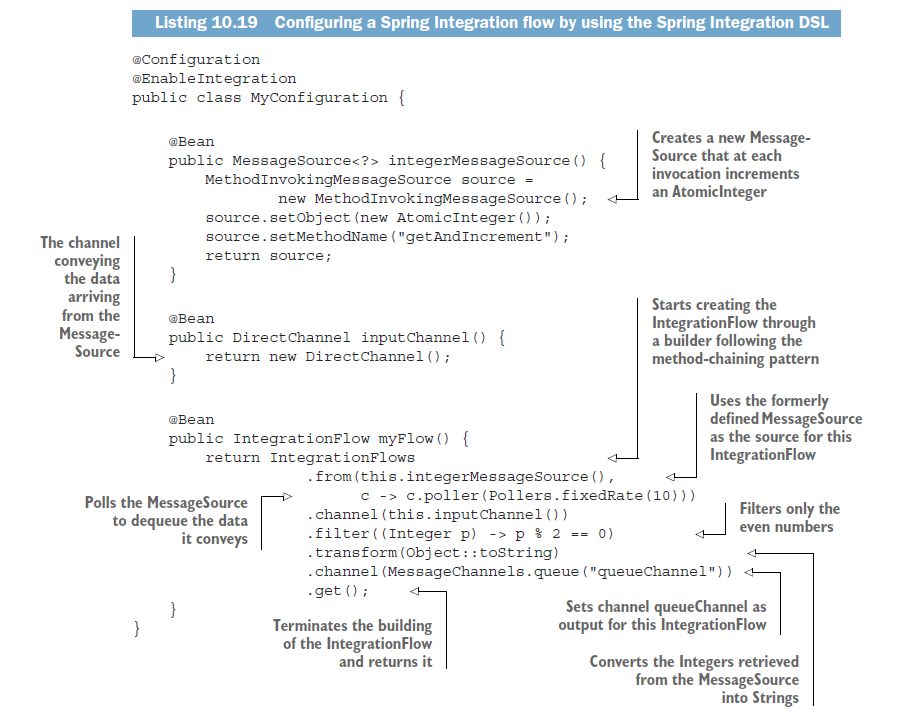
****

***Spring Integration***

*Spring Integration* extends the dependency-injection-based Spring programming model to support the well-known Enterprise Integration Patterns.1 Spring Integration’s primary goals are to provide a simple model to implement complex enterprise integration solutions and to promote the adoption of an asynchronous, message-driven architecture.

Spring Integration enables lightweight remoting, messaging, and scheduling within Spring-based applications. These features are also available through a rich, fluent DSL that’s more than syntactic sugar built on top of traditional Spring XML configuration files.

**Spring Integration** enables lightweight remoting, messaging, and scheduling within Spring-based applications. These features are also available through a rich, f**luent DSL that’s more than syntactic sugar built on** top of traditional Spring XML configuration files.

****

**Here, the method myFlow() builds an IntegrationFlow by using the Spring Integration**

DSL. It uses the fluent builder provided by the IntegrationFlows class, which implements the method-chaining pattern. In this case, the resulting flow polls a MessageSource at a fixed rate, providing a sequence of Integers; filters the even ones and converts them to Strings, and finally sends the result to an output channel in a

style that’s similar to the native Java 8 Stream API.

*Everyday Java*

*British computer scientist Tony Hoare introduced null references back in 1965* while designing ALGOL W, one of the first typed programming languages with heapallocated records, later saying that he did so “simply because it was so easy to implement.” Despite his goal “to ensure that all use of references could be absolutely safe, with checking performed automatically by the compiler,” he decided to make an exception for null references because he thought that they were the most convenient way to model *the absence of a value*. After many years, he regretted this decision, calling it “my billion-dollar mistake.” We’ve all seen the effect. We examine a field of an object, perhaps to determine whether its value is one of two expected forms, only to find that we’re examining not an object but a null pointer that promptly raises that annoying NullPointerException.

In reality, Hoare’s statement could underestimate the costs incurred by millions of developers fixing bugs caused by null references in the past 50 years.

***How do you model the absence of a value?***

Imagine that you have the following nested object structure for a person who owns a car and has car insurance in the following listing.

What’s problematic with the following code?

public String getCarInsuranceName(Person person) {

return person.getCar().getInsurance().getName();

}

This code looks pretty reasonable, but many people don’t own a car, so what’s the result of calling the method getCar? A common unfortunate practice is to return the null reference to indicate the absence of a value (here, to indicate the absence of a car). As a consequence, the call to getInsurance returns the insurance of a null reference,

which results in a NullPointerException at runtime and stops your program from running further. But that’s not all. What if person was null? What if the method getInsurance returned null too?

***Reducing NullPointerExceptions with defensive checking***

What can you do to avoid running into an unexpected NullPointerException? Typically, you can add null checks where necessary (and sometimes, in an excess of defensive programming, even where not necessary) and often with different styles.

***Reducing NullPointerExceptions with defensive checking***

What can you do to avoid running into an unexpected NullPointerException? Typically, you can add null checks where necessary (and sometimes, in an excess of defensive programming, even where not necessary) and often with different styles.

public String getCarInsuranceName(Person person) {

**//Each null check increases the nesting level of the remaining part of the invocation chain.**

if (person != null) {

Car car = person.getCar();

if (car != null) {

Insurance insurance = car.getInsurance();

if (insurance != null) {

return insurance.getName();

}

}

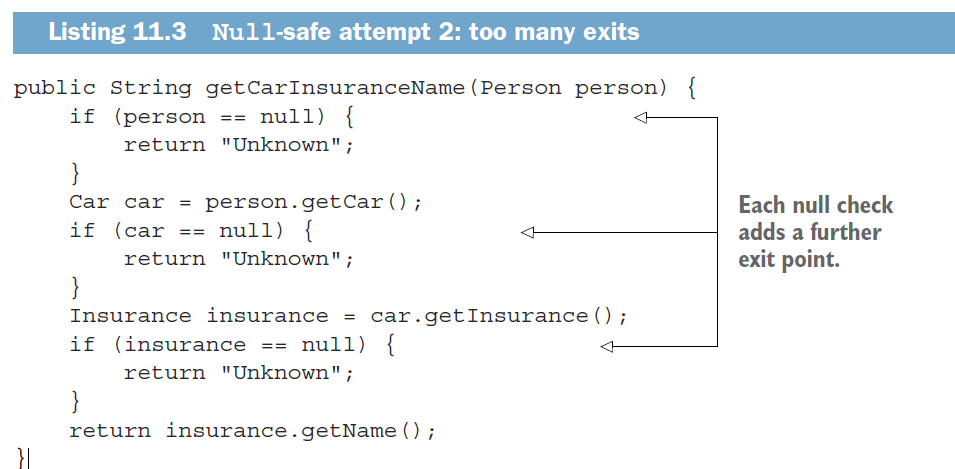
}

return "Unknown";

}

it shows a recurring

pattern: every time you doubt that a variable could be null, you’re obliged to add a further nested if block, increasing the indentation level of the code. This technique clearly scales poorly and compromises readability, so maybe you’d like to attempt another solution. Try to avoid this problem by doing something different as shown in the next listing.



In this second attempt, you try to avoid the deeply nested if blocks, adopting a different strategy: every time you meet a null variable, you return the string "Unknown". Nevertheless, this solution is also far from ideal; now the method has four distinct exit points, making it hard to maintain.

***Problems with null***

To recap our discussion so far, the use of null references in Java causes both theoretical

and practical problems:

 *It’s a source of error*. NullPointerException is by far the most common exception in Java.

 *It bloats your code*. It worsens readability by making it necessary to fill your code with null checks that are often deeply nested.

 *It’s meaningless*. It doesn’t have any semantic meaning, and in particular, it represents the wrong way to model the absence of a value in a statically typed language.

 *It breaks Java philosophy*. Java always hides pointers from developers except in one case: the null pointer.

 *It creates a hole in the type system*. null carries no type or other information, so it can be assigned to any reference type. This situation is a problem because when null is propagated to another part of the system, you have no idea what that null was initially supposed to be.

***What are the alternatives to null in other languages?***

In recent years, languages such as Groovy worked around this problem by introducing a *safe navigation operator*, represented by ?., to safely navigate potentially null values.

Or Spring has Elvis Presley operator shortened-ternary operator

To understand how this process works, consider the following **Groovy code**, which retrieves the name of the insurance company used by a given person to insure a car:

def carInsuranceName = person?.car?.insurance?.name

A similar feature was proposed and then discarded for Java 7. Somehow, though, we don’t seem to miss a safe navigation operator in Java. The first temptation of all Java developers when confronted with a NullPointerException is to fix it quickly by adding an if statement, checking that a value isn’t null before invoking a method on it.

Other functional languages, such as **Haskell** and **Scala**, take a different view. Haskell includes a Maybe type, which essentially encapsulates an optional value. A value of type Maybe can contain a value of a given type or nothing. Haskell no concept of a null reference. Scala has a similar construct called **Option**[T] to encapsulate the presence

or absence of a value of type T. **Java 8 takes inspiration from this idea** of an optional value by introducing a new class called java.util.**Optional**<T>!

***Introducing the Optional class***

Java 8 introduces a new class called java.util.Optional<T> that’s inspired by Haskell and Scala.

An important, practical semantic difference in using Optionals instead of nulls is that in the first case, declaring a variable of type Optional<Car> instead of Car clearly signals that a missing value is permitted there.

EMPTY OPTIONAL

As mentioned earlier, you can get hold of an empty optional object by using the static factory method Optional.empty: Optional<Car> optCar = Optional.empty();

OPTIONAL FROM A NON-NULL VALUE

You can also create an optional from a non-null value with the static factory method Optional.of:

Optional<Car> optCar = Optional.of(car);

If car were null, a NullPointerException would be thrown immediately (rather than

getting a latent error when you try to access properties of the car).

OPTIONAL FROM NULL

Finally, by using the static factory method Optional.ofNullable, you can create an Optional object that may hold a null value: Optional<Car> optCar = Optional.ofNullable(car);

If car were null, the resulting Optional object would be empty.

***Extracting and transforming values from Optionals with map***

A common pattern is to extract information from an object. You may want to extract the name from an insurance company, for example.

List<Dish> menu2 = Arrays.*asList*(**new** Dish(**null**, **false**, 800, Dish.Type.***MEAT***),

**new** Dish("salmon", **false**, 450, Dish.Type.***FISH***));

//null value is still mapped

List<String> dishNames3 = menu2.stream().map(Dish::getName).collect(*toList*());

System.***out***.println(dishNames3);

//This method is conceptually similar to the map method of Stream

Dish d = **new** Dish(**null**, **false**, 800, Dish.Type.***MEAT***);

Optional<Dish> optInsurance = Optional.*ofNullable*(d);

Optional<String> name = optInsurance.map(Dish::getName);//If the Optional is empty, nothing happens.

System.***out***.println(name); //Optional.empty

***Chaining Optional objects with flatMap***

Because you’ve learned how to use map, your first reaction may be to use map to rewrite

the code as follows:

Optional<Person> optPerson = Optional.of(person);

Optional<String> name =

optPerson.map(Person::getCar)

.map(Car::getInsurance)

.map(Insurance::getName);

Unfortunately, this code **doesn’t compile**. Why? The variable optPerson is of type Optional<Person>, so it’s perfectly fine to call the map method. But getCar returns an object of type Optional<Car> which means that the result of the map operation is an object of type **Optional<Optional<Car>>.**

How can you solve this problem? Again, you can look at a pattern you’ve used previously with streams: the flatMap method.

public String getCarInsuranceName(Optional<Person> person) {

return person.flatMap(Person::getCar)

.flatMap(Car::getInsurance)

.map(Insurance::getName)

.orElse("Unknown");

}

Using optionals in a domain model and why they’re not serializable

In listing 11.4, we showed how to use Optionals in your domain model to mark with a specific type the values that are allowed to be missing or remain undefined. The designers of the Optional class, however, developed it based on different assumptions and with a different use case in mind. In particular, Java language architect

Brian Goetz clearly stated that the purpose of Optional is to support the optionalreturn idiom only.

Because the Optional class wasn’t intended for use as a field type, it doesn’t implement the Serializable interface. For this reason, using Optionals in your domain model could break applications with tools or frameworks that require a serializable model to work.

Alternatively, **if you need to have a serializable domain model**, we suggest that you at least provide a method allowing access to any possibly missing value as an optional, as in the following example:

public class Person {

private Car car;

public Optional<Car> getCarAsOptional() {

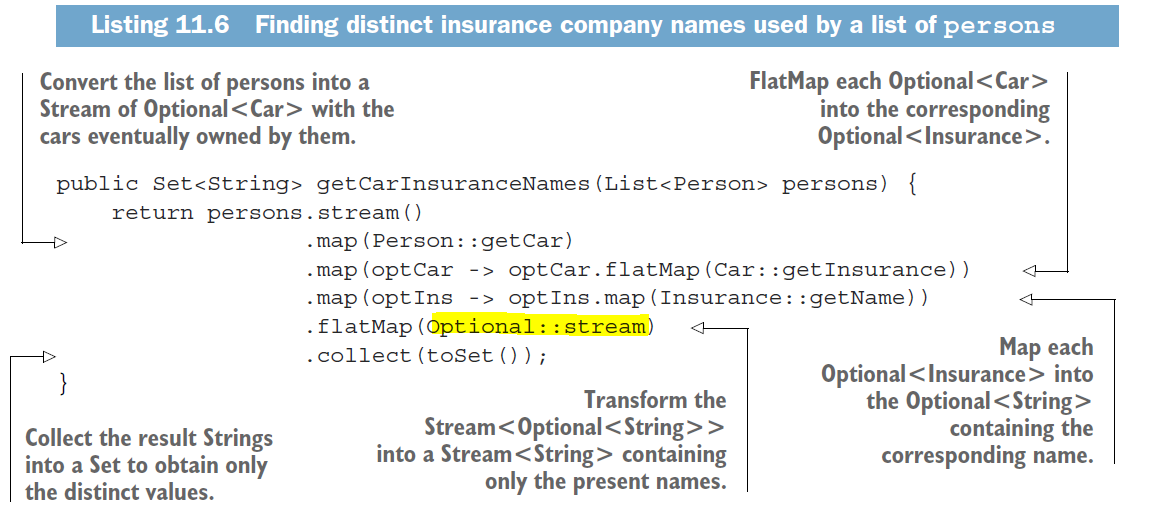
return Optional.ofNullable(car);

}

}

***Manipulating a stream of optionals***

The Optional’s stream() method, introduced in Java 9,



public String getCarInsuranceName(Optional<Person> person, int minAge) {

return person.**filter**(p -> p.getAge() >= minAge)

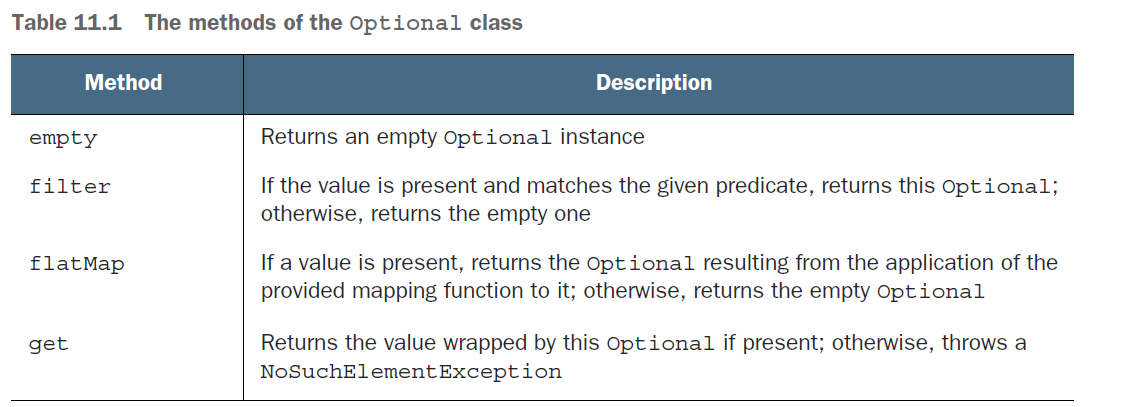
.flatMap(Person::getCar)

.flatMap(Car::getInsurance)

.map(Insurance::getName)

.orElse("Unknown");

}



***Primitive optionals and why you shouldn’t use them***

Note that like streams, optionals also have primitive counterparts—OptionalInt, OptionalLong, and OptionalDouble—so the method in listing 11.7 could have returned OptionalInt instead of Optional<Integer>.

***Summary***

 null references were historically introduced in programming languages to signal

the absence of a value.

 Java 8 introduced the class java.util.Optional<T> to model the presence or

absence of a value.

 You can create Optional objects with the static factory methods Optional.empty,

Optional.of, and Optional.ofNullable.

 The Optional class supports many methods—such as map, flatMap, and filter—

that are conceptually similar to the methods of a stream.

 Using Optional forces you to actively unwrap an optional to deal with the

absence of a value; as a result, you protect your code against unintended null

pointer exceptions.

 Using Optional can help you design better APIs in which, by reading the signature

of a method, users can tell whether to expect an optional value.

*New Date and Time API*

*In Java 1.0, the only support for date and time was the* ***java.util.Date*** *class.* Despite its name, this class **doesn’t represent a date**, but a **point in time with millisecond precision**. Even worse, the usability of this class is harmed by some nebulous design decisions such as the choice of its offsets: the **years start from 1900**, whereas the months start at index 0, day started with 1. If you wanted to represent the release date of Java 9, which is 21 September 2017, you’d have to create an instance of Date as follows:

Date date = new Date(117, 8, 21);

Printing this date produces, for the authors: Thu Sep 21 00:00:00 CET 2017 Not very intuitive, is it? Moreover, even the String returned by the toString method of the Date class could be quite **misleading**.

As a consequence, in Java 1.1 many methods of the Date class were deprecated, and the class was replaced by the alternative java.util. Calendar class. Unfortunately, Calendar has similar problems and design flaws that

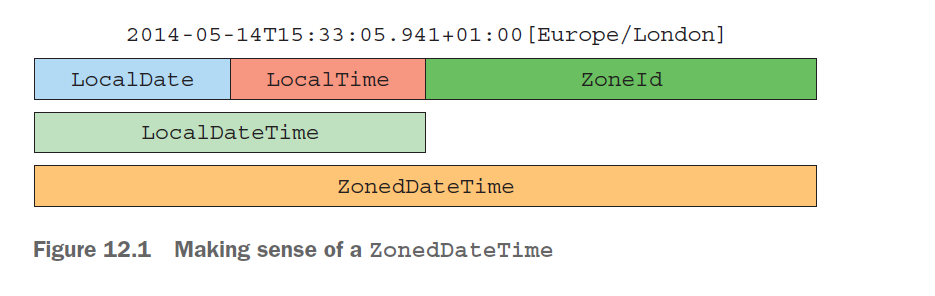
lead to error-prone code. Months also start at index 0. (At least Calendar got rid of the 1900 offset for the year.) Worse, the presence of *both* the Date and Calendar classes increases confusion among developers.

The DateFormat comes with its own set of problems. It isn’t thread-safe, for example, which means that if two threads try to parse a date by using the same formatter at the same time, you may receive unpredictable results.

The consequence is that all these flaws and inconsistencies have encouraged the use of **third-party** date and time libraries, such as **Joda-Time** (I tried in 2011 at Commerzbank). For these reasons, Oracle decided to provide high-quality date and time support in the native Java API. As a result, Java 8 integrates many of the Joda-Time features in the **java.time** package.

***LocalDate, LocalTime, LocalDateTime, Instant,***

***Duration, and Period, ZonedDateTime, ZoneId, ZonedRegion, ..***



ISLAMIC CALENDAR

Of the new calendars added to Java 8, the HijrahDate (Islamic calendar) seems to be the most complex because it can have variants. The Hijrah calendar system is based on lunar months. There are a variety of methods to determine a new month, such as a new moon that could be visible anywhere in the world or that must be visible first in Saudi Arabia. The withVariant method is used to choose the desired variant. Java 8 includes the Umm Al-Qura variant for HijrahDate as standard. The following code illustrates an example of displaying the start and end dates of

Ramadan for the current Islamic year in ISO date:

***Summary***

 The old java.util.Date class and all other classes used to model dates and times in Java before Java 8 have many inconsistencies and design flaws, including mutability and some poorly chosen offsets, defaults, and naming.

 All the date-time objects of the new Date and Time API are immutable.

 This new API provides two different time representations to manage the different needs of humans and machines when operating on it.

 You can manipulate date and time objects in both an absolute and relative manner, and the result of these manipulations is always a new instance, leaving the original one unchanged.

 TemporalAdjusters allow you to manipulate a date in a more complex way than changing one of its values, and you can define and use your own custom date transformations.

 You can define a formatter to print and parse date-time objects in a specific format. These formatters can be created from a pattern or programmatically, and they’re all thread-safe.

 You can represent a time zone, relative to a specific region/location and as a fixed offset from UTC/Greenwich, and apply it to a date-time object to localize it.

 You can use calendar systems different from the ISO-8601 standard system.

Different types of compatibilities: binary, source, and behavioral

There are **three main kinds of compatibility** when introducing a change to a Java program: **binary**, **source**, and **behavioral compatibilities** (see https://blogs.oracle.com/darcy/entry/kinds\_of\_compatibility). You saw that adding a method to an interface is binary compatible but results in a compiler error if the class implementing the interface is recompiled. It’s good to know the different kinds of compatibilities, so in this sidebar, we examine them in detail. ***Binary compatibility* means** that existing binaries running without errors continue to link (which involves verification, preparation, and resolution) without error after introducing a change. Adding a method to an interface is binary compatible, for example, because if it’s not called, existing methods of the interface can still run without problems. In its simplest form, ***source compatibility* means** that an existing program will still compile after introducing a change. Adding a method to an interface isn’t source compatible; existing implementations won’t recompile because they need to implement the new method.

Finally, *b****ehavioral compatibility* means** running a program after a change with the same input results in the same behavior. Adding a method to an interface is behavioral compatible because the method is never called in the program (or gets overridden by an implementation).

***Three resolution rules to know***

You have three rules to follow when a class inherits a method with the same signature

from multiple places (such as another class or interface):

1 Classes always win. A method declaration in the class or a superclass takes priority over any default method declaration.

2 Otherwise, subinterfaces win: the method with the same signature in the most specific default-providing interface is selected. (If B extends A, B is more specific than A.)

3 Finally, if the choice is still ambiguous, the class inheriting from multiple interfaces has to explicitly select which default method implementation to use by overriding it and calling the desired method explicitly.

You’ve seen that the default method’s resolution mechanism is simple if a class inherits from several methods with the same signature. Follow three rules systematically to solve all possible conflicts:

1 First, an explicit method declaration in the class or a superclass takes priority over any default method declaration.

2 Otherwise, the method with the same signature in the most specific defaultproviding interface is selected.

3 Finally, if there’s still a conflict, you have to explicitly override the default methods and choose which one your class should use.

***Summary***

 Interfaces in Java 8 can have implementation code through default methods and static methods.

 Default methods start with a default keyword and contain a body, as class methods do.

 Adding an abstract method to a published interface is a source incompatibility.

 Default methods help library designers evolve APIs in a backward-compatible way.

 Default methods can be used for creating optional methods and multiple inheritance of behavior.

 Resolution rules exist to resolve conflicts when a class inherits from several default methods with the same signature.

 A method declaration in the class or a superclass takes priority over any default method declaration. Otherwise, the method with the same signature in the most specific default-providing interface is selected.

 When two methods are equally specific, a class must explicitly override this method, such as to select which one to call.

*The Java Module System*

***Separation of concerns***

Separation of concerns (SoC) is a principle that promotes decomposing a computer program into distinct features. Suppose that you need to develop an accounting application that parses expenses **in different formats**, analyzes them, and provides summary reports to your customer. By **applying SoC**, you split parsing, analysis, and reporting into separate parts called *modules*—cohesive groups of code that have little overlap. In other words, a module groups classes, allowing you to express visibility relationships between classes in your application.

*The SoC principle is useful at an architectural point of view (such as model versus* view versus controller) and in a low-level approach (such as separating the business logic from the recovery mechanism). The benefits are

 Allowing work on individual parts in isolation, which helps team collaboration

 Facilitating reuse of separate parts

 Easier maintenance of the overall system

***Information hiding***

*Information hiding* is a principle that encourages hiding implementation details. You often hear the term ***encapsulation* used** to indicate that a specific piece of code is so well isolated from the other parts of the application that changing its internal implementation won’t negatively affect them.

*You make your code modular by grouping packages, classes, and* interfaces that address a specific concern.

LIMITED VISIBILITY CONTROL

As discussed in the previous section, Java provides access modifiers to support information hiding. These modifiers are public, protected, package-level, and private visibility. But what about controlling visibility *between* packages?

CLASS PATH

Earlier in this chapter, we discussed the benefits of software written in a way that makes it simple to maintain and understand—in other words, easier to reason about. We also talked about separation of concerns and modeling dependencies between modules. Unfortunately, Java historically falls short in supporting these ideas when it comes to bundling and running an application. In fact, you have to ship all your compiled classes into one single flat JAR, which is made accessible from the class path. Then the JVM can dynamically locate and load classes from the class path as needed.

**Second**, the class path doesn’t support explicit dependencies; all the classes inside different JARs are merged into one bag of classes on the class path. In other words, the class path doesn’t let you declare explicitly that one JAR depends on a set of classes contained inside another JAR. This situation makes it difficult to reason about the class path and to ask questions such as:

 Is anything missing?

 Are there any conflicts?

Build tools such as Maven and Gradle can help you solve this problem. Before Java 9, however, neither Java nor the JVM had any support for explicit dependencies. The issues combined are often referred to as JAR Hell or Class Path Hell.

***Monolithic JDK***

The *Java Development Kit* (JDK) is a collection of tools that lets you work with and run Java programs.

How can you get away from this problem as a whole ecosystem? Java 8 introduced the notion of *compact profiles* as a step forward. Three profiles were introduced to have different memory footprints, depending on which parts of the JDK library you’re interested in. Compact profiles, however, provided only a short-term fix. Many internal APIs in the JDK aren’t meant for public use.

Unfortunately, due to the poor encapsulation provided by the Java language, those APIs are commonly used. The

class **sun.misc.Unsafe**, for example, is used by several libraries (including Spring, Netty, and Mockito) but was never intended to be made available outside the JDK internals. As a result, it’s extremely difficult to evolve these APIs without introducing incompatible changes.

***Comparison with OSGi***

This section compares Java 9 modules with OSGi. If you haven’t heard of OSGi, we suggest that you skip this section.

Before the introduction of modules based on project Jigsaw into Java 9, Java already had a powerful module system, named OSGi, even if it wasn’t formally part of the Java platform. The Open Service Gateway initiative (**OSGi**) started in 2000 and, until the arrival of Java 9, represented the de-facto standard for implementing a modular application on the JVM.

In reality, OSGi and the new Java 9 Module System aren’t mutually exclusive; they can coexist in the same application. In fact, their features overlap only partially. OSGi has a much wider scope and provides many capabilities that aren’t available in **Jigsaw**.

OSGi modules are called *bundles* and run inside a specific OSGi framework. Several certified OSGi framework implementations exist, but the two with the widest adoption are Apache Felix and Equinox (which is also used to run the Eclipse IDE).

***Java modules: the big picture***

Java 9 provides a new unit of Java program structure: the *module*. A module is introduced with a new keyword2 module, followed by its name and its body. Such a *module descriptor*3 lives in a special file: **module-info.java**, which is compiled to **module-info.class** .

***Fine-grained and coarse-grained modularization***

When you’re modularizing a system, you can choose the granularity. In the most **finegrained** scheme, every package has its own module (as in the previous section); in the most **coarse-grained** scheme, a single module contains all the packages in your system.

***The exports clause***

Here’s how we might declare the module expenses.readers. (Don’t worry about the syntax and concepts yet; we cover these topics later.)

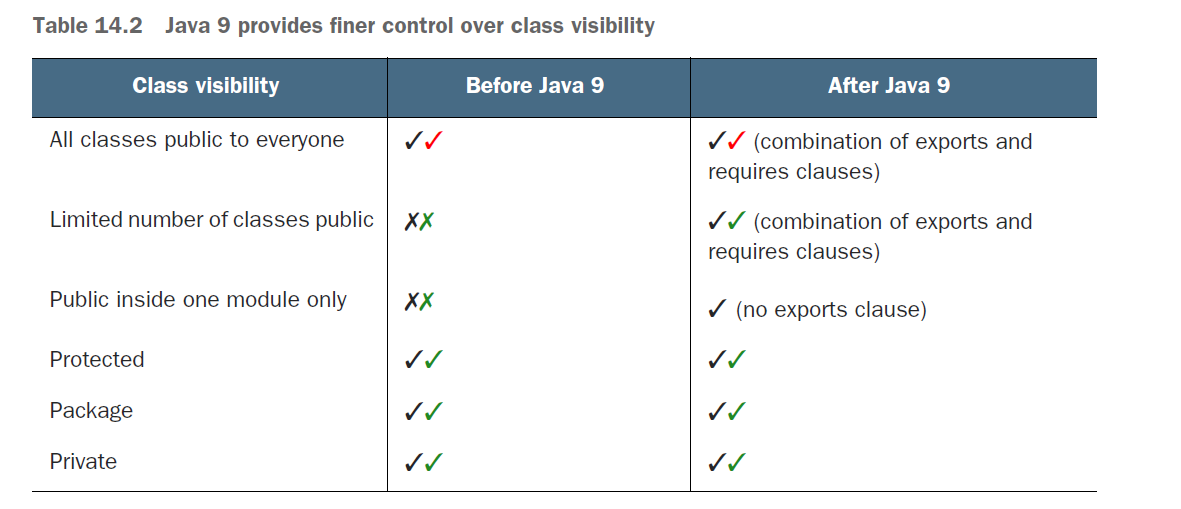
module expenses.readers {

//**These are package names, not module names.**

exports com.example.expenses.readers;

**requires** java.base; //**This is a package name, not a module name.**

}



***Automatic modules***

You may decide that the implementation of your HttpReader is low-level; instead, you’d like to use a specialized library such as the httpclient from the Apache project. How do you incorporate that library into your project? You’ve learned about the

requires clause, so try to add it in the module-info.java for the expenses.readers project. Run mvn clean package again to see what happens. Unfortunately, the result is bad news: [ERROR] module not found: httpclient