CHAPTER 7

Lambdas and Streams

In Java 8, functional interfaces, lambdas, and method references were added to make it easier to create function objects. The streams API was added in tandem with these language changes to provide library support for processing sequences of data elements. In this chapter, we discuss how to make best use of these facilities.

Item 42: Prefer lambdas to anonymous classes

Historically, interfaces (or, rarely, abstract classes) with a single abstract method were used as *function types*. Their instances, known as *function objects*, represent functions or actions. Since JDK 1.1 was released in 1997, the primary means of creating a function object was the *anonymous class* (Item 24). Here's a code snippet to sort a list of strings in order of length, using an anonymous class to create the sort's comparison function (which imposes the sort order):

```
// Anonymous class instance as a function object - obsolete!
Collections.sort(words, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});
```

Anonymous classes were adequate for the classic objected-oriented design patterns requiring function objects, notably the *Strategy* pattern [Gamma95]. The `Comparator` interface represents an *abstract strategy* for sorting; the anonymous class above is a *concrete strategy* for sorting strings. The verbosity of anonymous classes, however, made functional programming in Java an unappealing prospect.

In Java 8, the language formalized the notion that interfaces with a single abstract method are special and deserve special treatment. These interfaces are now known as *functional interfaces*, and the language allows you to create instances of these interfaces using *lambda expressions*, or *lambdas* for short.

Lambdas are similar in function to anonymous classes, but far more concise. Here's how the code snippet above looks with the anonymous class replaced by a lambda. The boilerplate is gone, and the behavior is clearly evident:

```
// Lambda expression as function object (replaces anonymous class)
Collections.sort(words,
        (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Note that the types of the lambda (`Comparator<String>`), of its parameters (`s1` and `s2`, both `String`), and of its return value (`int`) are not present in the code. The compiler deduces these types from context, using a process known as *type inference*. In some cases, the compiler won't be able to determine the types, and you'll have to specify them. The rules for type inference are complex: they take up an entire chapter in the JLS [JLS, 18]. Few programmers understand these rules in detail, but that's OK. **Omit the types of all lambda parameters unless their presence makes your program clearer.** If the compiler generates an error telling you it can't infer the type of a lambda parameter, *then* specify it. Sometimes you may have to cast the return value or the entire lambda expression, but this is rare.

One caveat should be added concerning type inference. Item 26 tells you not to use raw types, Item 29 tells you to favor generic types, and Item 30 tells you to favor generic methods. This advice is doubly important when you're using lambdas, because the compiler obtains most of the type information that allows it to perform type inference from generics. If you don't provide this information, the compiler will be unable to do type inference, and you'll have to specify types manually in your lambdas, which will greatly increase their verbosity. By way of example, the code snippet above won't compile if the variable `words` is declared to be of the raw type `List` instead of the parameterized type `List<String>`.

Incidentally, the comparator in the snippet can be made even more succinct if a *comparator construction method* is used in place of a lambda (Items 14. 43):

```
Collections.sort(words, comparingInt(String::length));
```

In fact, the snippet can be made still shorter by taking advantage of the `sort` method that was added to the `List` interface in Java 8:

```
words.sort(comparingInt(String::length));
```

The addition of lambdas to the language makes it practical to use function objects where it would not previously have made sense. For example, consider the `Operation` enum type in Item 34. Because each enum required different behavior for its `apply` method, we used constant-specific class bodies and overrode the `apply` method in each enum constant. To refresh your memory, here is the code:

```
// Enum type with constant-specific class bodies & data (Item 34)
public enum Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };

    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    @Override public String toString() { return symbol; }

    public abstract double apply(double x, double y);
}
```

Item 34 says that enum instance fields are preferable to constant-specific class bodies. Lambdas make it easy to implement constant-specific behavior using the former instead of the latter. Merely pass a lambda implementing each enum constant's behavior to its constructor. The constructor stores the lambda in an instance field, and the `apply` method forwards invocations to the lambda. The resulting code is simpler and clearer than the original version:

```
// Enum with function object fields & constant-specific behavior
public enum Operation {
    PLUS  ("+", (x, y) -> x + y),
    MINUS ("-", (x, y) -> x - y),
    TIMES ("*", (x, y) -> x * y),
    DIVIDE("/", (x, y) -> x / y);

    private final String symbol;
    private final DoubleBinaryOperator op;

    Operation(String symbol, DoubleBinaryOperator op) {
        this.symbol = symbol;
        this.op = op;
    }

    @Override public String toString() { return symbol; }

    public double apply(double x, double y) {
        return op.applyAsDouble(x, y);
    }
}
```

Note that we're using the `DoubleBinaryOperator` interface for the lambdas that represent the enum constant's behavior. This is one of the many predefined functional interfaces in `java.util.function` (Item 44). It represents a function that takes two `double` arguments and returns a `double` result.

Looking at the lambda-based `Operation` enum, you might think constant-specific method bodies have outlived their usefulness, but this is not the case. Unlike methods and classes, **lambdas lack names and documentation; if a computation isn't self-explanatory, or exceeds a few lines, don't put it in a lambda.** One line is ideal for a lambda, and three lines is a reasonable maximum. If you violate this rule, it can cause serious harm to the readability of your programs. If a lambda is long or difficult to read, either find a way to simplify it or refactor your program to eliminate it. Also, the arguments passed to enum constructors are evaluated in a static context. Thus, lambdas in enum constructors can't access instance members of the enum. Constant-specific class bodies are still the way to go if an enum type has constant-specific behavior that is difficult to understand, that can't be implemented in a few lines, or that requires access to instance fields or methods.

Likewise, you might think that anonymous classes are obsolete in the era of lambdas. This is closer to the truth, but there are a few things you can do with anonymous classes that you can't do with lambdas. Lambdas are limited to functional interfaces. If you want to create an instance of an abstract class, you can do it with an anonymous class, but not a lambda. Similarly, you can use anonymous classes to create instances of interfaces with multiple abstract methods. Finally, a lambda cannot obtain a reference to itself. In a lambda, the `this` keyword refers to the enclosing instance, which is typically what you want. In an anonymous class, the `this` keyword refers to the anonymous class instance. If you need access to the function object from within its body, then you must use an anonymous class.

Lambdas share with anonymous classes the property that you can't reliably serialize and deserialize them across implementations. Therefore, **you should rarely, if ever, serialize a lambda** (or an anonymous class instance). If you have a function object that you want to make serializable, such as a `Comparator`, use an instance of a private static nested class (Item 24).

In summary, as of Java 8, lambdas are by far the best way to represent small function objects. **Don't use anonymous classes for function objects unless you have to create instances of types that aren't functional interfaces.** Also, remember that lambdas make it so easy to represent small function objects that it opens the door to functional programming techniques that were not previously practical in Java.

Item 43: Prefer method references to lambdas

The primary advantage of lambdas over anonymous classes is that they are more succinct. Java provides a way to generate function objects even more succinct than lambdas: *method references*. Here is a code snippet from a program that maintains a map from arbitrary keys to `Integer` values. If the value is interpreted as a count of the number of instances of the key, then the program is a multiset implementation. The function of the code snippet is to associate the number 1 with the key if it is not in the map and to increment the associated value if the key is already present:

```
map.merge(key, 1, (count, incr) -> count + incr);
```

Note that this code uses the `merge` method, which was added to the `Map` interface in Java 8. If no mapping is present for the given key, the method simply inserts the given value; if a mapping is already present, `merge` applies the given function to the current value and the given value and overwrites the current value with the result. This code represents a typical use case for the `merge` method.

The code reads nicely, but there's still some boilerplate. The parameters `count` and `incr` don't add much value, and they take up a fair amount of space. Really, all the lambda tells you is that the function returns the sum of its two arguments. As of Java 8, `Integer` (and all the other boxed numerical primitive types) provides a static method `sum` that does exactly the same thing. We can simply pass a reference to this method and get the same result with less visual clutter:

```
map.merge(key, 1, Integer::sum);
```

The more parameters a method has, the more boilerplate you can eliminate with a method reference. In some lambdas, however, the parameter names you choose provide useful documentation, making the lambda more readable and maintainable than a method reference, even if the lambda is longer.

There's nothing you can do with a method reference that you can't also do with a lambda (with one obscure exception—see JLS, 9.9-2 if you're curious). That said, method references usually result in shorter, clearer code. They also give you an out if a lambda gets too long or complex: You can extract the code from the lambda into a new method and replace the lambda with a reference to that method. You can give the method a good name and document it to your heart's content.

If you're programming with an IDE, it will offer to replace a lambda with a method reference wherever it can. You should usually, but not always, take the IDE up on the offer. Occasionally, a lambda will be more succinct than a method reference. This happens most often when the method is in the same class as the

lambda. For example, consider this snippet, which is presumed to occur in a class named `GoshThisClassNameIsHumongous`:

```
service.execute(GoshThisClassNameIsHumongous::action);
```

The lambda equivalent looks like this:

```
service.execute(() -> action());
```

The snippet using the method reference is neither shorter nor clearer than the snippet using the lambda, so prefer the latter. Along similar lines, the `Function` interface provides a generic static factory method to return the identity function, `Function.identity()`. It's typically shorter and cleaner *not* to use this method but to code the equivalent lambda inline: `x -> x`.

Many method references refer to static methods, but there are four kinds that do not. Two of them are *bound* and *unbound* instance method references. In bound references, the receiving object is specified in the method reference. Bound references are similar in nature to static references: the function object takes the same arguments as the referenced method. In unbound references, the receiving object is specified when the function object is applied, via an additional parameter before the method's declared parameters. Unbound references are often used as mapping and filter functions in stream pipelines (Item 45). Finally, there are two kinds of *constructor* references, for classes and arrays. Constructor references serve as factory objects. All five kinds of method references are summarized in the table below:

Method Ref Type	Example	Lambda Equivalent
Static	`Integer::parseInt`	`str -> Integer.parseInt(str)`
Bound	`Instant.now()::isAfter`	`Instant then = Instant.now(); t -> then.isAfter(t)`
Unbound	`String::toLowerCase`	`str -> str.toLowerCase()`
Class Constructor	`TreeMap<K,V>::new`	`() -> new TreeMap<K,V>`
Array Constructor	`int[]::new`	`len -> new int[len]`

In summary, method references often provide a more succinct alternative to lambdas. **Where method references are shorter and clearer, use them; where they aren't, stick with lambdas.**

Item 44: Favor the use of standard functional interfaces

Now that Java has lambdas, best practices for writing APIs have changed considerably. For example, the *Template Method* pattern [Gamma95], wherein a subclass overrides a *primitive method* to specialize the behavior of its superclass, is far less attractive. The modern alternative is to provide a static factory or constructor that accepts a function object to achieve the same effect. More generally, you'll be writing more constructors and methods that take function objects as parameters. Choosing the right functional parameter type demands care.

Consider `LinkedHashMap`. You can use this class as a cache by overriding its protected `removeEldestEntry` method, which is invoked by `put` each time a new key is added to the map. When this method returns `true`, the map removes its eldest entry, which is passed to the method. The following override allows the map to grow to one hundred entries and then deletes the eldest entry each time a new key is added, maintaining the hundred most recent entries:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
   return size() > 100;
}
```

This technique works fine, but you can do much better with lambdas. If `LinkedHashMap` were written today, it would have a static factory or constructor that took a function object. Looking at the declaration for `removeEldestEntry`, you might think that the function object should take a `Map.Entry<K,V>` and return a `boolean`, but that wouldn't quite do it: The `removeEldestEntry` method calls `size()` to get the number of entries in the map, which works because `removeEldestEntry` is an instance method on the map. The function object that you pass to the constructor is not an instance method on the map and can't capture it because the map doesn't exist yet when its factory or constructor is invoked. Thus, the map must pass itself to the function object, which must therefore take the map on input as well as its eldest entry. If you were to declare such a functional interface, it would look something like this:

```
// Unnecessary functional interface; use a standard one instead.
@FunctionalInterface interface EldestEntryRemovalFunction<K,V>{
    boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
```

This interface would work fine, but you shouldn't use it, because you don't need to declare a new interface for this purpose. The `java.util.function` package provides a large collection of standard functional interfaces for your use.

If one of the standard functional interfaces does the job, you should generally use it in preference to a purpose-built functional interface. This will make your API easier to learn, by reducing its conceptual surface area, and will provide significant interoperability benefits, as many of the standard functional interfaces provide useful default methods. The `Predicate` interface, for instance, provides methods to combine predicates. In the case of our `LinkedHashMap` example, the standard `BiPredicate<Map<K,V>, Map.Entry<K,V>>` interface should be used in preference to a custom `EldestEntryRemovalFunction` interface.

There are forty-three interfaces in `java.util.Function`. You can't be expected to remember them all, but if you remember six basic interfaces, you can derive the rest when you need them. The basic interfaces operate on object reference types. The `Operator` interfaces represent functions whose result and argument types are the same. The `Predicate` interface represents a function that takes an argument and returns a `boolean`. The `Function` interface represents a function whose argument and return types differ. The `Supplier` interface represents a function that takes no arguments and returns (or "supplies") a value. Finally, `Consumer` represents a function that takes an argument and returns nothing, essentially consuming its argument. The six basic functional interfaces are summarized below:

Interface	Function Signature	Example
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Instant::now`
`Consumer<T>`	`void accept(T t)`	`System.out::println`

There are also three variants of each of the six basic interfaces to operate on the primitive types `int`, `long`, and `double`. Their names are derived from the basic interfaces by prefixing them with a primitive type. So, for example, a predicate that takes an `int` is an `IntPredicate`, and a binary operator that takes two `long` values and returns a `long` is a `LongBinaryOperator`. None of these variant types is parameterized except for the `Function` variants, which are parameterized by return type. For example, `LongFunction<int[]>` takes a `long` and returns an `int[]`.

There are nine additional variants of the `Function` interface, for use when the result type is primitive. The source and result types always differ, because a function from a type to itself is a `UnaryOperator`. If both the source and result types are primitive, prefix `Function` with *Src*`To`*Result*, for example `LongToIntFunction` (six variants). If the source is a primitive and the result is an object reference, prefix `Function` with `<Src>ToObj`, for example `DoubleToObjFunction` (three variants).

There are two-argument versions of the three basic functional interfaces for which it makes sense to have them: `BiPredicate<T,U>`, `BiFunction<T,U,R>`, and `BiConsumer<T,U>`. There are also `BiFunction` variants returning the three relevant primitive types: `ToIntBiFunction<T,U>`, `ToLongBiFunction<T,U>`, and `ToDoubleBiFunction<T,U>`. There are two-argument variants of `Consumer` that take one object reference and one primitive type: `ObjDoubleConsumer<T>`, `ObjIntConsumer<T>`, and `ObjLongConsumer<T>`. In total, there are nine two-argument versions of the basic interfaces.

Finally, there is the `BooleanSupplier` interface, a variant of `Supplier` that returns `boolean` values. This is the only explicit mention of the `boolean` type in any of the standard functional interface names, but `boolean` return values are supported via `Predicate` and its four variant forms. The `BooleanSupplier` interface and the forty-two interfaces described in the previous paragraphs account for all forty-three standard functional interfaces. Admittedly, this is a lot to swallow, and not terribly orthogonal. On the other hand, the bulk of the functional interfaces that you'll need have been written for you and their names are regular enough that you shouldn't have too much trouble coming up with one when you need it.

Most of the standard functional interfaces exist only to provide support for primitive types. **Don't be tempted to use basic functional interfaces with boxed primitives instead of primitive functional interfaces.** While it works, it violates the advice of Item 61, "prefer primitive types to boxed primitives." The performance consequences of using boxed primitives for bulk operations can be deadly.

Now you know that you should typically use standard functional interfaces in preference to writing your own. But when *should* you write your own? Of course you need to write your own if none of the standard ones does what you need, for example if you require a predicate that takes three parameters, or one that throws a checked exception. But there are times you should write your own functional interface even when one of the standard ones is structurally identical.

Consider our old friend `Comparator<T>`, which is structurally identical to the `ToIntBiFunction<T,T>` interface. Even if the latter interface had existed when the former was added to the libraries, it would have been wrong to use it. There

are several reasons that `Comparator` deserves its own interface. First, its name provides excellent documentation every time it is used in an API, and it's used a lot. Second, the `Comparator` interface has strong requirements on what constitutes a valid instance, which comprise its *general contract*. By implementing the interface, you are pledging to adhere to its contract. Third, the interface is heavily outfitted with useful default methods to transform and combine comparators.

You should seriously consider writing a purpose-built functional interface in preference to using a standard one if you need a functional interface that shares one or more of the following characteristics with `Comparator`:

- It will be commonly used and could benefit from a descriptive name.
- It has a strong contract associated with it.
- It would benefit from custom default methods.

If you elect to write your own functional interface, remember that it's an interface and hence should be designed with great care (Item 21).

Notice that the `EldestEntryRemovalFunction` interface (page 199) is labeled with the `@FunctionalInterface` annotation. This annotation type is similar in spirit to `@Override`. It is a statement of programmer intent that serves three purposes: it tells readers of the class and its documentation that the interface was designed to enable lambdas; it keeps you honest because the interface won't compile unless it has exactly one abstract method; and it prevents maintainers from accidentally adding abstract methods to the interface as it evolves. **Always annotate your functional interfaces with the `@FunctionalInterface` annotation.**

A final point should be made concerning the use of functional interfaces in APIs. Do not provide a method with multiple overloadings that take different functional interfaces in the same argument position if it could create a possible ambiguity in the client. This is not just a theoretical problem. The `submit` method of `ExecutorService` can take either a `Callable<T>` or a `Runnable`, and it is possible to write a client program that requires a cast to indicate the correct overloading (Item 52). The easiest way to avoid this problem is not to write overloadings that take different functional interfaces in the same argument position. This is a special case of the advice in Item 52, "use overloading judiciously."

In summary, now that Java has lambdas, it is imperative that you design your APIs with lambdas in mind. Accept functional interface types on input and return them on output. It is generally best to use the standard interfaces provided in `java.util.function.Function`, but keep your eyes open for the relatively rare cases where you would be better off writing your own functional interface.

Item 45: Use streams judiciously

The streams API was added in Java 8 to ease the task of performing bulk operations, sequentially or in parallel. This API provides two key abstractions: the *stream*, which represents a finite or infinite sequence of data elements, and the *stream pipeline*, which represents a multistage computation on these elements. The elements in a stream can come from anywhere. Common sources include collections, arrays, files, regular expression pattern matchers, pseudorandom number generators, and other streams. The data elements in a stream can be object references or primitive values. Three primitive types are supported: `int`, `long`, and `double`.

A stream pipeline consists of a source stream followed by zero or more *intermediate operations* and one *terminal operation*. Each intermediate operation transforms the stream in some way, such as mapping each element to a function of that element or filtering out all elements that do not satisfy some condition. Intermediate operations all transform one stream into another, whose element type may be the same as the input stream or different from it. The terminal operation performs a final computation on the stream resulting from the last intermediate operation, such as storing its elements into a collection, returning a certain element, or printing all of its elements.

Stream pipelines are evaluated *lazily*: evaluation doesn't start until the terminal operation is invoked, and data elements that aren't required in order to complete the terminal operation are never computed. This lazy evaluation is what makes it possible to work with infinite streams. Note that a stream pipeline without a terminal operation is a silent no-op, so don't forget to include one.

The streams API is *fluent*: it is designed to allow all of the calls that comprise a pipeline to be chained into a single expression. In fact, multiple pipelines can be chained together into a single expression.

By default, stream pipelines run sequentially. Making a pipeline execute in parallel is as simple as invoking the `parallel` method on any stream in the pipeline, but it is seldom appropriate to do so (Item 48).

The streams API is sufficiently versatile that practically any computation can be performed using streams, but just because you can doesn't mean you should. When used appropriately, streams can make programs shorter and clearer; when used inappropriately, they can make programs difficult to read and maintain. There are no hard and fast rules for when to use streams, but there are heuristics.

Consider the following program, which reads the words from a dictionary file and prints all the anagram groups whose size meets a user-specified minimum. Recall that two words are anagrams if they consist of the same letters in a different

order. The program reads each word from a user-specified dictionary file and places the words into a map. The map key is the word with its letters alphabetized, so the key for "staple" is "aelpst", and the key for "petals" is also "aelpst": the two words are anagrams, and all anagrams share the same alphabetized form (or *alphagram*, as it is sometimes known). The map value is a list containing all of the words that share an alphabetized form. After the dictionary has been processed, each list is a complete anagram group. The program then iterates through the map's values() view and prints each list whose size meets the threshold:

```
// Prints all large anagram groups in a dictionary iteratively
public class Anagrams {
    public static void main(String[] args) throws IOException {
        File dictionary = new File(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        Map<String, Set<String>> groups = new HashMap<>();
        try (Scanner s = new Scanner(dictionary)) {
            while (s.hasNext()) {
                String word = s.next();
                groups.computeIfAbsent(alphabetize(word),
                    (unused) -> new TreeSet<>()).add(word);
            }
        }

        for (Set<String> group : groups.values())
            if (group.size() >= minGroupSize)
                System.out.println(group.size() + ": " + group);
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

One step in this program is worthy of note. The insertion of each word into the map, which is shown in bold, uses the computeIfAbsent method, which was added in Java 8. This method looks up a key in the map: If the key is present, the method simply returns the value associated with it. If not, the method computes a value by applying the given function object to the key, associates this value with the key, and returns the computed value. The computeIfAbsent method simplifies the implementation of maps that associate multiple values with each key.

Now consider the following program, which solves the same problem, but makes heavy use of streams. Note that the entire program, with the exception of

the code that opens the dictionary file, is contained in a single expression. The only reason the dictionary is opened in a separate expression is to allow the use of the `try`-with-resources statement, which ensures that the dictionary file is closed:

```
// Overuse of streams - don't do this!
public class Anagrams {
  public static void main(String[] args) throws IOException {
    Path dictionary = Paths.get(args[0]);
    int minGroupSize = Integer.parseInt(args[1]);

      try (Stream<String> words = Files.lines(dictionary)) {
        words.collect(
          groupingBy(word -> word.chars().sorted()
                      .collect(StringBuilder::new,
                        (sb, c) -> sb.append((char) c),
                        StringBuilder::append).toString()))
          .values().stream()
            .filter(group -> group.size() >= minGroupSize)
            .map(group -> group.size() + ": " + group)
            .forEach(System.out::println);
        }
    }
}
```

If you find this code hard to read, don't worry; you're not alone. It is shorter, but it is also less readable, especially to programmers who are not experts in the use of streams. **Overusing streams makes programs hard to read and maintain.**

Luckily, there is a happy medium. The following program solves the same problem, using streams without overusing them. The result is a program that's both shorter and clearer than the original:

```
// Tasteful use of streams enhances clarity and conciseness
public class Anagrams {
   public static void main(String[] args) throws IOException {
      Path dictionary = Paths.get(args[0]);
      int minGroupSize = Integer.parseInt(args[1]);

      try (Stream<String> words = Files.lines(dictionary)) {
         words.collect(groupingBy(word -> alphabetize(word)))
           .values().stream()
           .filter(group -> group.size() >= minGroupSize)
           .forEach(g -> System.out.println(g.size() + ": " + g));
      }
   }

   // alphabetize method is the same as in original version
}
```

Even if you have little previous exposure to streams, this program is not hard to understand. It opens the dictionary file in a `try`-with-resources block, obtaining a stream consisting of all the lines in the file. The `stream` variable is named `words` to suggest that each element in the stream is a word. The pipeline on this stream has no intermediate operations; its terminal operation collects all the words into a map that groups the words by their alphabetized form (Item 46). This is exactly the same map that was constructed in both previous versions of the program. Then a new `Stream<List<String>>` is opened on the `values()` view of the map. The elements in this stream are, of course, the anagram groups. The stream is filtered so that all of the groups whose size is less than `minGroupSize` are ignored, and finally, the remaining groups are printed by the terminal operation `forEach`.

Note that the lambda parameter names were chosen carefully. The parameter `g` should really be named `group`, but the resulting line of code would be too wide for the book. **In the absence of explicit types, careful naming of lambda parameters is essential to the readability of stream pipelines.**

Note also that word alphabetization is done in a separate `alphabetize` method. This enhances readability by providing a name for the operation and keeping implementation details out of the main program. **Using helper methods is even more important for readability in stream pipelines than in iterative code** because pipelines lack explicit type information and named temporary variables.

The `alphabetize` method could have been reimplemented to use streams, but a stream-based `alphabetize` method would have been less clear, more difficult to write correctly, and probably slower. These deficiencies result from Java's lack of support for primitive `char` streams (which is not to imply that Java should have supported `char` streams; it would have been infeasible to do so). To demonstrate the hazards of processing `char` values with streams, consider the following code:

```
"Hello world!".chars().forEach(System.out::print);
```

You might expect it to print `Hello world!`, but if you run it, you'll find that it prints `721011081081113211911111410810033`. This happens because the elements of the stream returned by `"Hello world!".chars()` are not `char` values but `int` values, so the `int` overloading of `print` is invoked. It is admittedly confusing that a method named `chars` returns a stream of `int` values. You *could* fix the program by using a cast to force the invocation of the correct overloading:

```
"Hello world!".chars().forEach(x -> System.out.print((char) x));
```

but ideally you should **refrain from using streams to process `char` values.**

When you start using streams, you may feel the urge to convert all your loops into streams, but resist the urge. While it may be possible, it will likely harm the readability and maintainability of your code base. As a rule, even moderately complex tasks are best accomplished using some combination of streams and iteration, as illustrated by the `Anagrams` programs above. So **refactor existing code to use streams and use them in new code only where it makes sense to do so.**

As shown in the programs in this item, stream pipelines express repeated computation using function objects (typically lambdas or method references), while iterative code expresses repeated computation using code blocks. There are some things you can do from code blocks that you can't do from function objects:

- From a code block, you can read or modify any local variable in scope; from a lambda, you can only read final or effectively final variables [JLS 4.12.4], and you can't modify any local variables.
- From a code block, you can `return` from the enclosing method, `break` or `continue` an enclosing loop, or throw any checked exception that this method is declared to throw; from a lambda you can do none of these things.

If a computation is best expressed using these techniques, then it's probably not a good match for streams. Conversely, streams make it very easy to do some things:

- Uniformly transform sequences of elements
- Filter sequences of elements
- Combine sequences of elements using a single operation (for example to add them, concatenate them, or compute their minimum)
- Accumulate sequences of elements into a collection, perhaps grouping them by some common attribute
- Search a sequence of elements for an element satisfying some criterion

If a computation is best expressed using these techniques, then it is a good candidate for streams.

One thing that is hard to do with streams is to access corresponding elements from multiple stages of a pipeline simultaneously: once you map a value to some other value, the original value is lost. One workaround is to map each value to a *pair object* containing the original value and the new value, but this is not a satisfying solution, especially if the pair objects are required for multiple stages of a pipeline. The resulting code is messy and verbose, which defeats a primary purpose of streams. When it is applicable, a better workaround is to invert the mapping when you need access to the earlier-stage value.

For example, let's write a program to print the first twenty *Mersenne primes*. To refresh your memory, a *Mersenne number* is a number of the form $2^p - 1$. If p is prime, the corresponding Mersenne number *may* be prime; if so, it's a Mersenne prime. As the initial stream in our pipeline, we want all the prime numbers. Here's a method to return that (infinite) stream. We assume a static import has been used for easy access to the static members of `BigInteger`:

```
static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

The name of the method (`primes`) is a plural noun describing the elements of the stream. This naming convention is highly recommended for all methods that return streams because it enhances the readability of stream pipelines. The method uses the static factory `Stream.iterate`, which takes two parameters: the first element in the stream, and a function to generate the next element in the stream from the previous one. Here is the program to print the first twenty Mersenne primes:

```
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}
```

This program is a straightforward encoding of the prose description above: it starts with the primes, computes the corresponding Mersenne numbers, filters out all but the primes (the magic number `50` controls the probabilistic primality test), limits the resulting stream to twenty elements, and prints them out.

Now suppose that we want to precede each Mersenne prime with its exponent (p). This value is present only in the initial stream, so it is inaccessible in the terminal operation, which prints the results. Luckily, it's easy to compute the exponent of a Mersenne number by inverting the mapping that took place in the first intermediate operation. The exponent is simply the number of bits in the binary representation, so this terminal operation generates the desired result:

```
.forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
```

There are plenty of tasks where it is not obvious whether to use streams or iteration. For example, consider the task of initializing a new deck of cards. Assume that `Card` is an immutable value class that encapsulates a `Rank` and a `Suit`, both of which are enum types. This task is representative of any task that

requires computing all the pairs of elements that can be chosen from two sets. Mathematicians call this the *Cartesian product* of the two sets. Here's an iterative implementation with a nested for-each loop that should look very familiar to you:

```
// Iterative Cartesian product computation
private static List<Card> newDeck() {
    List<Card> result = new ArrayList<>();
    for (Suit suit : Suit.values())
        for (Rank rank : Rank.values())
            result.add(new Card(suit, rank));
    return result;
}
```

And here is a stream-based implementation that makes use of the intermediate operation `flatMap`. This operation maps each element in a stream to a stream and then concatenates all of these new streams into a single stream (or *flattens* them). Note that this implementation contains a nested lambda, shown in boldface:

```
// Stream-based Cartesian product computation
private static List<Card> newDeck() {
    return Stream.of(Suit.values())
        .flatMap(suit ->
            Stream.of(Rank.values())
                .map(rank -> new Card(suit, rank)))
        .collect(toList());
}
```

Which of the two versions of `newDeck` is better? It boils down to personal preference and the environment in which you're programming. The first version is simpler and perhaps feels more natural. A larger fraction of Java programmers will be able to understand and maintain it, but some programmers will feel more comfortable with the second (stream-based) version. It's a bit more concise and not too difficult to understand if you're reasonably well-versed in streams and functional programming. If you're not sure which version you prefer, the iterative version is probably the safer choice. If you prefer the stream version and you believe that other programmers who will work with the code will share your preference, then you should use it.

In summary, some tasks are best accomplished with streams, and others with iteration. Many tasks are best accomplished by combining the two approaches. There are no hard and fast rules for choosing which approach to use for a task, but there are some useful heuristics. In many cases, it will be clear which approach to use; in some cases, it won't. **If you're not sure whether a task is better served by streams or iteration, try both and see which works better.**

Item 46: Prefer side-effect-free functions in streams

If you're new to streams, it can be difficult to get the hang of them. Merely expressing your computation as a stream pipeline can be hard. When you succeed, your program will run, but you may realize little if any benefit. Streams isn't just an API, it's a paradigm based on functional programming. In order to obtain the expressiveness, speed, and in some cases parallelizability that streams have to offer, you have to adopt the paradigm as well as the API.

The most important part of the streams paradigm is to structure your computation as a sequence of transformations where the result of each stage is as close as possible to a *pure function* of the result of the previous stage. A pure function is one whose result depends only on its input: it does not depend on any mutable state, nor does it update any state. In order to achieve this, any function objects that you pass into stream operations, both intermediate and terminal, should be free of side-effects.

Occasionally, you may see streams code that looks like this snippet, which builds a frequency table of the words in a text file:

```
// Uses the streams API but not the paradigm--Don't do this!
Map<String, Long> freq = new HashMap<>();
try (Stream<String> words = new Scanner(file).tokens()) {
    words.forEach(word -> {
        freq.merge(word.toLowerCase(), 1L, Long::sum);
    });
}
```

What's wrong with this code? After all, it uses streams, lambdas, and method references, and gets the right answer. Simply put, it's not streams code at all; it's iterative code masquerading as streams code. It derives no benefits from the streams API, and it's (a bit) longer, harder to read, and less maintainable than the corresponding iterative code. The problem stems from the fact that this code is doing all its work in a terminal `forEach` operation, using a lambda that mutates external state (the frequency table). A `forEach` operation that does anything more than present the result of the computation performed by a stream is a "bad smell in code," as is a lambda that mutates state. So how should this code look?

```
// Proper use of streams to initialize a frequency table
Map<String, Long> freq;
try (Stream<String> words = new Scanner(file).tokens()) {
    freq = words
        .collect(groupingBy(String::toLowerCase, counting()));
}
```

This snippet does the same thing as the previous one but makes proper use of the streams API. It's shorter and clearer. So why would anyone write it the other way? Because it uses tools they're already familiar with. Java programmers know how to use for-each loops, and the `forEach` terminal operation is similar. But the `forEach` operation is among the least powerful of the terminal operations and the least stream-friendly. It's explicitly iterative, and hence not amenable to parallelization. **The `forEach` operation should be used only to report the result of a stream computation, not to perform the computation.** Occasionally, it makes sense to use `forEach` for some other purpose, such as adding the results of a stream computation to a preexisting collection.

The improved code uses a *collector*, which is a new concept that you have to learn in order to use streams. The `Collectors` API is intimidating: it has thirty-nine methods, some of which have as many as five type parameters. The good news is that you can derive most of the benefit from this API without delving into its full complexity. For starters, you can ignore the `Collector` interface and think of a collector as an opaque object that encapsulates a *reduction* strategy. In this context, reduction means combining the elements of a stream into a single object. The object produced by a collector is typically a collection (which accounts for the name collector).

The collectors for gathering the elements of a stream into a true `Collection` are straightforward. There are three such collectors: `toList()`, `toSet()`, and `toCollection(collectionFactory)`. They return, respectively, a set, a list, and a programmer-specified collection type. Armed with this knowledge, we can write a stream pipeline to extract a top-ten list from our frequency table.

```
// Pipeline to get a top-ten list of words from a frequency table
List<String> topTen = freq.keySet().stream()
    .sorted(comparing(freq::get).reversed())
    .limit(10)
    .collect(toList());
```

Note that we haven't qualified the `toList` method with its class, `Collectors`. **It is customary and wise to statically import all members of `Collectors` because it makes stream pipelines more readable.**

The only tricky part of this code is the comparator that we pass to `sorted`, `comparing(freq::get).reversed()`. The `comparing` method is a comparator construction method (Item 14) that takes a key extraction function. The function takes a word, and the "extraction" is actually a table lookup: the bound method reference `freq::get` looks up the word in the frequency table and returns the number of times the word appears in the file. Finally, we call `reversed` on the

comparator, so we're sorting the words from most frequent to least frequent. Then it's a simple matter to limit the stream to ten words and collect them into a list.

The previous code snippets use `Scanner`'s `stream` method to get a stream over the scanner. This method was added in Java 9. If you're using an earlier release, you can translate the scanner, which implements `Iterator`, into a stream using an adapter similar to the one in Item 47 (`streamOf(Iterable<E>)`).

So what about the other thirty-six methods in `Collectors`? Most of them exist to let you collect streams into maps, which is far more complicated than collecting them into true collections. Each stream element is associated with a key *and a value*, and multiple stream elements can be associated with the same key.

The simplest map collector is `toMap(keyMapper, valueMapper)`, which takes two functions, one of which maps a stream element to a key, the other, to a value. We used this collector in our `fromString` implementation in Item 34 to make a map from the string form of an enum to the enum itself:

```
// Using a toMap collector to make a map from string to enum
private static final Map<String, Operation> stringToEnum =
    Stream.of(values()).collect(
        toMap(Object::toString, e -> e));
```

This simple form of `toMap` is perfect if each element in the stream maps to a unique key. If multiple stream elements map to the same key, the pipeline will terminate with an `IllegalStateException`.

The more complicated forms of `toMap`, as well as the `groupingBy` method, give you various ways to provide strategies for dealing with such collisions. One way is to provide the `toMap` method with a *merge function* in addition to its key and value mappers. The merge function is a `BinaryOperator<V>`, where `V` is the value type of the map. Any additional values associated with a key are combined with the existing value using the merge function, so, for example, if the merge function is multiplication, you end up with a value that is the product of all the values associated with the key by the value mapper.

The three-argument form of `toMap` is also useful to make a map from a key to a chosen element associated with that key. For example, suppose we have a stream of record albums by various artists, and we want a map from recording artist to best-selling album. This collector will do the job.

```
// Collector to generate a map from key to chosen element for key
Map<Artist, Album> topHits = albums.collect(
   toMap(Album::artist, a->a, maxBy(comparing(Album::sales))));
```

Note that the comparator uses the static factory method `maxBy`, which is statically imported from `BinaryOperator`. This method converts a `Comparator<T>` into a `BinaryOperator<T>` that computes the maximum implied by the specified comparator. In this case, the comparator is returned by the comparator construction method `comparing`, which takes the key extractor function `Album::sales`. This may seem a bit convoluted, but the code reads nicely. Loosely speaking, it says, "convert the stream of albums to a map, mapping each artist to the album that has the best album by sales." This is surprisingly close to the problem statement.

Another use of the three-argument form of `toMap` is to produce a collector that imposes a last-write-wins policy when there are collisions. For many streams, the results will be nondeterministic, but if all the values that may be associated with a key by the mapping functions are identical, or if they are all acceptable, this collector's s behavior may be just what you want:

```
// Collector to impose last-write-wins policy
toMap(keyMapper, valueMapper, (v1, v2) -> v2)
```

The third and final version of `toMap` takes a fourth argument, which is a map factory, for use when you want to specify a particular map implementation such as an `EnumMap` or a `TreeMap`.

There are also variant forms of the first three versions of `toMap`, named `toConcurrentMap`, that run efficiently in parallel and produce `ConcurrentHashMap` instances.

In addition to the `toMap` method, the `Collectors` API provides the `groupingBy` method, which returns collectors to produce maps that group elements into categories based on a *classifier function*. The classifier function takes an element and returns the category into which it falls. This category serves as the element's map key. The simplest version of the `groupingBy` method takes only a classifier and returns a map whose values are lists of all the elements in each category. This is the collector that we used in the `Anagram` program in Item 45 to generate a map from alphabetized word to a list of the words sharing the alphabetization:

```
words.collect(groupingBy(word -> alphabetize(word)))
```

If you want `groupingBy` to return a collector that produces a map with values other than lists, you can specify a *downstream collector* in addition to a classifier. A downstream collector produces a value from a stream containing all the

elements in a category. The simplest use of this parameter is to pass `toSet()`, which results in a map whose values are sets of elements rather than lists.

Alternatively, you can pass `toCollection(collectionFactory)`, which lets you create the collections into which each category of elements is placed. This gives you the flexibility to choose any collection type you want. Another simple use of the two-argument form of `groupingBy` is to pass `counting()` as the downstream collector. This results in a map that associates each category with the *number* of elements in the category, rather than a collection containing the elements. That's what you saw in the frequency table example at the beginning of this item:

```
Map<String, Long> freq = words
        .collect(groupingBy(String::toLowerCase, counting()));
```

The third version of `groupingBy` lets you specify a map factory in addition to a downstream collector. Note that this method violates the standard telescoping argument list pattern: the `mapFactory` parameter precedes, rather than follows, the `downStream` parameter. This version of `groupingBy` gives you control over the containing map as well as the contained collections, so, for example, you can specify a collector that returns a `TreeMap` whose values are `TreeSets`.

The `groupingByConcurrent` method provides variants of all three overloadings of `groupingBy`. These variants run efficiently in parallel and produce `ConcurrentHashMap` instances. There is also a rarely used relative of `groupingBy` called `partitioningBy`. In lieu of a classifier method, it takes a predicate and returns a map whose key is a `Boolean`. There are two overloadings of this method, one of which takes a downstream collector in addition to a predicate.

The collectors returned by the `counting` method are intended *only* for use as downstream collectors. The same functionality is available directly on `Stream`, via the `count` method, so **there is never a reason to say `collect(counting())`**. There are fifteen more `Collectors` methods with this property. They include the nine methods whose names begin with `summing`, `averaging`, and `summarizing` (whose functionality is available on the corresponding primitive stream types). They also include all overloadings of the `reducing` method, and the `filtering`, `mapping`, `flatMapping`, and `collectingAndThen` methods. Most programmers can safely ignore the majority of these methods. From a design perspective, these collectors represent an attempt to partially duplicate the functionality of streams in collectors so that downstream collectors can act as "ministreams."

There are three `Collectors` methods we have yet to mention. Though they are in `Collectors`, they don't involve collections. The first two are `minBy` and `maxBy`, which take a comparator and return the minimum or maximum element in

the stream as determined by the comparator. They are minor generalizations of the `min` and `max` methods in the `Stream` interface and are the collector analogues of the binary operators returned by the like-named methods in `BinaryOperator`. Recall that we used `BinaryOperator.maxBy` in our best-selling album example.

The final `Collectors` method is `joining`, which operates only on streams of `CharSequence` instances such as strings. In its parameterless form, it returns a collector that simply concatenates the elements. Its one argument form takes a single `CharSequence` parameter named `delimiter` and returns a collector that joins the stream elements, inserting the delimiter between adjacent elements. If you pass in a comma as the delimiter, the collector returns a comma-separated values string (but beware that the string will be ambiguous if any of the elements in the stream contain commas). The three argument form takes a prefix and suffix in addition to the delimiter. The resulting collector generates strings like the ones that you get when you print a collection, for example `[came, saw, conquered]`.

In summary, the essence of programming stream pipelines is side-effect-free function objects. This applies to all of the many function objects passed to streams and related objects. The terminal operation `forEach` should only be used to report the result of a computation performed by a stream, not to perform the computation. In order to use streams properly, you have to know about collectors. The most important collector factories are `toList`, `toSet`, `toMap`, `groupingBy`, and `joining`.

Item 47: Prefer Collection to Stream as a return type

Many methods return sequences of elements. Prior to Java 8, the obvious return types for such methods were the collection interfaces `Collection`, `Set`, and `List`; `Iterable`; and the array types. Usually, it was easy to decide which of these types to return. The norm was a collection interface. If the method existed solely to enable for-each loops or the returned sequence couldn't be made to implement some `Collection` method (typically, `contains(Object)`), the `Iterable` interface was used. If the returned elements were primitive values or there were stringent performance requirements, arrays were used. In Java 8, streams were added to the platform, substantially complicating the task of choosing the appropriate return type for a sequence-returning method.

You may hear it said that streams are now the obvious choice to return a sequence of elements, but as discussed in Item 45, streams do not make iteration obsolete: writing good code requires combining streams and iteration judiciously. If an API returns only a stream and some users want to iterate over the returned sequence with a for-each loop, those users will be justifiably upset. It is especially frustrating because the `Stream` interface contains the sole abstract method in the `Iterable` interface, and `Stream`'s specification for this method is compatible with `Iterable`'s. The only thing preventing programmers from using a for-each loop to iterate over a stream is `Stream`'s failure to extend `Iterable`.

Sadly, there is no good workaround for this problem. At first glance, it might appear that passing a method reference to `Stream`'s `iterator` method would work. The resulting code is perhaps a bit noisy and opaque, but not unreasonable:

```
// Won't compile, due to limitations on Java's type inference
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
    // Process the process
}
```

Unfortunately, if you attempt to compile this code, you'll get an error message:

```
Test.java:6: error: method reference not expected here
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
                        ^
```

In order to make the code compile, you have to cast the method reference to an appropriately parameterized `Iterable`:

```
// Hideous workaround to iterate over a stream
for (ProcessHandle ph : (Iterable<ProcessHandle>)
                         ProcessHandle.allProcesses()::iterator)
```

This client code works, but it is too noisy and opaque to use in practice. A better workaround is to use an adapter method. The JDK does not provide such a method, but it's easy to write one, using the same technique used in-line in the snippets above. Note that no cast is necessary in the adapter method because Java's type inference works properly in this context:

```
// Adapter from  Stream<E> to Iterable<E>
public static <E> Iterable<E> iterableOf(Stream<E> stream) {
    return stream::iterator;
}
```

With this adapter, you can iterate over any stream with a for-each statement:

```
for (ProcessHandle p : iterableOf(ProcessHandle.allProcesses())) {
    // Process the process
}
```

Note that the stream versions of the `Anagrams` program in Item 34 use the `Files.lines` method to read the dictionary, while the iterative version uses a scanner. The `Files.lines` method is superior to a scanner, which silently swallows any exceptions encountered while reading the file. Ideally, we would have used `Files.lines` in the iterative version too. This is the sort of compromise that programmers will make if an API provides only stream access to a sequence and they want to iterate over the sequence with a for-each statement.

Conversely, a programmer who wants to process a sequence using a stream pipeline will be justifiably upset by an API that provides only an `Iterable`. Again the JDK does not provide an adapter, but it's easy enough to write one:

```
// Adapter from Iterable<E> to Stream<E>
public static <E> Stream<E> streamOf(Iterable<E> iterable) {
    return StreamSupport.stream(iterable.spliterator(), false);
}
```

If you're writing a method that returns a sequence of objects and you know that it will only be used in a stream pipeline, then of course you should feel free to return a stream. Similarly, a method returning a sequence that will only be used for iteration should return an `Iterable`. But if you're writing a public API that returns a sequence, you should provide for users who want to write stream pipelines as well as those who want to write for-each statements, unless you have a good reason to believe that most of your users will want to use the same mechanism.

The `Collection` interface is a subtype of `Iterable` and has a `stream` method, so it provides for both iteration and stream access. Therefore, **`Collection` or an appropriate subtype is generally the best return type for a public, sequence-returning method.** Arrays also provide for easy iteration and stream access with the `Arrays.asList` and `Stream.of` methods. If the sequence you're returning is small enough to fit easily in memory, you're probably best off returning one of the standard collection implementations, such as `ArrayList` or `HashSet`. But **do not store a large sequence in memory just to return it as a collection.**

If the sequence you're returning is large but can be represented concisely, consider implementing a special-purpose collection. For example, suppose you want to return the *power set* of a given set, which consists of all of its subsets. The power set of {*a*, *b*, *c*} is {{}, {*a*}, {*b*}, {*c*}, {*a*, *b*}, {*a, c*}, {*b*, *c*}, {*a*, *b*, *c*}}. If a set has *n* elements, its power set has 2^n. Therefore, you shouldn't even consider storing the power set in a standard collection implementation. It is, however, easy to implement a custom collection for the job with the help of `AbstractList`.

The trick is to use the index of each element in the power set as a bit vector, where the *n*th bit in the index indicates the presence or absence of the *n*th element from the source set. In essence, there is a natural mapping between the binary numbers from 0 to $2^n - 1$ and the power set of an *n*-element set. Here's the code:

```
// Returns the power set of an input set as custom collection
public class PowerSet {
   public static final <E> Collection<Set<E>> of(Set<E> s) {
      List<E> src = new ArrayList<>(s);
      if (src.size() > 30)
         throw new IllegalArgumentException("Set too big " + s);
      return new AbstractList<Set<E>>() {
         @Override public int size() {
            return 1 << src.size(); // 2 to the power srcSize
         }

         @Override public boolean contains(Object o) {
            return o instanceof Set && src.containsAll((Set)o);
         }

         @Override public Set<E> get(int index) {
            Set<E> result = new HashSet<>();
            for (int i = 0; index != 0; i++, index >>= 1)
               if ((index & 1) == 1)
                  result.add(src.get(i));
            return result;
         }
      };
   }
}
```

Note that `PowerSet.of` throws an exception if the input set has more than 30 elements. This highlights a disadvantage of using `Collection` as a return type rather than `Stream` or `Iterable`: `Collection` has an `int`-returning `size` method, which limits the length of the returned sequence to `Integer.MAX_VALUE`, or $2^{31} - 1$. The `Collection` specification does allow the `size` method to return $2^{31} - 1$ if the collection is larger, even infinite, but this is not a wholly satisfying solution.

In order to write a `Collection` implementation atop `AbstractCollection`, you need implement only two methods beyond the one required for `Iterable`: `contains` and `size`. Often it's easy to write efficient implementations of these methods. If it isn't feasible, perhaps because the contents of the sequence aren't predetermined before iteration takes place, return a stream or iterable, whichever feels more natural. If you choose, you can return both using two separate methods.

There are times when you'll choose the return type based solely on ease of implementation. For example, suppose you want to write a method that returns all of the (contiguous) sublists of an input list. It takes only three lines of code to generate these sublists and put them in a standard collection, but the memory required to hold this collection is quadratic in the size of the source list. While this is not as bad as the power set, which is exponential, it is clearly unacceptable. Implementing a custom collection, as we did for the power set, would be tedious, more so because the JDK lacks a skeletal `Iterator` implementation to help us.

It is, however, straightforward to implement a stream of all the sublists of an input list, though it does require a minor insight. Let's call a sublist that contains the first element of a list a *prefix* of the list. For example, the prefixes of (*a*, *b*, *c*) are (*a*), (*a*, *b*), and (*a*, *b*, *c*). Similarly, let's call a sublist that contains the last element a *suffix*, so the suffixes of (*a*, *b*, *c*) are (*a*, *b*, *c*), (*b*, c), and (*c*). The insight is that the sublists of a list are simply the suffixes of the prefixes (or identically, the prefixes of the suffixes) and the empty list. This observation leads directly to a clear, reasonably concise implementation:

```
// Returns a stream of all the sublists of its input list
public class SubLists {
   public static <E> Stream<List<E>> of(List<E> list) {
      return Stream.concat(Stream.of(Collections.emptyList()),
         prefixes(list).flatMap(SubLists::suffixes));
   }

   private static <E> Stream<List<E>> prefixes(List<E> list) {
      return IntStream.rangeClosed(1, list.size())
         .mapToObj(end -> list.subList(0, end));
   }
```

```
    private static <E> Stream<List<E>> suffixes(List<E> list) {
        return IntStream.range(0, list.size())
            .mapToObj(start -> list.subList(start, list.size()));
    }
}
```

Note that the `Stream.concat` method is used to add the empty list into the returned stream. Also note that the `flatMap` method (Item 45) is used to generate a single stream consisting of all the suffixes of all the prefixes. Finally, note that we generate the prefixes and suffixes by mapping a stream of consecutive `int` values returned by `IntStream.range` and `IntStream.rangeClosed`. This idiom is, roughly speaking, the stream equivalent of the standard `for`-loop on integer indices. Thus, our sublist implementation is similar in spirit to the obvious nested `for`-loop:

```
for (int start = 0; start < src.size(); start++)
    for (int end = start + 1; end <= src.size(); end++)
        System.out.println(src.subList(start, end));
```

It is possible to translate this `for`-loop directly into a stream. The result is more concise than our previous implementation, but perhaps a bit less readable. It is similar in spirit to the streams code for the Cartesian product in Item 45:

```
// Returns a stream of all the sublists of its input list
public static <E> Stream<List<E>> of(List<E> list) {
    return IntStream.range(0, list.size())
        .mapToObj(start ->
            IntStream.rangeClosed(start + 1, list.size())
                .mapToObj(end -> list.subList(start, end)))
        .flatMap(x -> x);
}
```

Like the `for`-loop that precedes it, this code does *not* emit the empty list. In order to fix this deficiency, you could either use `concat`, as we did in the previous version, or replace `1` by `(int) Math.signum(start)` in the `rangeClosed` call.

Either of these stream implementations of sublists is fine, but both will require some users to employ a `Stream`-to-`Iterable` adapter or to use a stream in places where iteration would be more natural. Not only does the `Stream`-to-`Iterable` adapter clutter up client code, but it slows down the loop by a factor of 2.3 on my machine. A purpose-built `Collection` implementation (not shown here) is considerably more verbose but runs about 1.4 times as fast as our stream-based implementation on my machine.

In summary, when writing a method that returns a sequence of elements, remember that some of your users may want to process them as a stream while others may want to iterate over them. Try to accommodate both groups. If it's feasible to return a collection, do so. If you already have the elements in a collection or the number of elements in the sequence is small enough to justify creating a new one, return a standard collection such as `ArrayList`. Otherwise, consider implementing a custom collection as we did for the power set. If it isn't feasible to return a collection, return a stream or iterable, whichever seems more natural. If, in a future Java release, the `Stream` interface declaration is modified to extend `Iterable`, then you should feel free to return streams because they will allow for both stream processing and iteration.

Item 48: Use caution when making streams parallel

Among mainstream languages, Java has always been at the forefront of providing facilities to ease the task of concurrent programming. When Java was released in 1996, it had built-in support for threads, with synchronization and `wait`/`notify`. Java 5 introduced the `java.util.concurrent` library, with concurrent collections and the executor framework. Java 7 introduced the fork-join package, a high-performance framework for parallel decomposition. Java 8 introduced streams, which can be parallelized with a single call to the `parallel` method. Writing concurrent programs in Java keeps getting easier, but writing concurrent programs that are correct and fast is as difficult as it ever was. Safety and liveness violations are a fact of life in concurrent programming, and parallel stream pipelines are no exception.

Consider this program from Item 45:

```
// Stream-based program to generate the first 20 Mersenne primes
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}

static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

On my machine, this program immediately starts printing primes and takes 12.5 seconds to run to completion. Suppose I naively try to speed it up by adding a call to `parallel()` to the stream pipeline. What do you think will happen to its performance? Will it get a few percent faster? A few percent slower? Sadly, what happens is that it doesn't print anything, but CPU usage spikes to 90 percent and stays there indefinitely (a *liveness failure*). The program might terminate eventually, but I was unwilling to find out; I stopped it forcibly after half an hour.

What's going on here? Simply put, the streams library has no idea how to parallelize this pipeline and the heuristics fail. Even under the best of circumstances, **parallelizing a pipeline is unlikely to increase its performance if the source is from `Stream.iterate`, or the intermediate operation `limit` is used.** This pipeline has to contend with *both* of these issues. Worse, the default parallelization strategy deals with the unpredictability of `limit` by assuming there's no harm in processing a few extra elements and discarding any unneeded results. In this case,

it takes roughly twice as long to find each Mersenne prime as it did to find the previous one. Thus, the cost of computing a single extra element is roughly equal to the cost of computing all previous elements combined, and this innocuous-looking pipeline brings the automatic parallelization algorithm to its knees. The moral of this story is simple: **Do not parallelize stream pipelines indiscriminately.** The performance consequences may be disastrous.

As a rule, **performance gains from parallelism are best on streams over `ArrayList`, `HashMap`, `HashSet`, and `ConcurrentHashMap` instances; arrays; `int` ranges; and `long` ranges.** What these data structures have in common is that they can all be accurately and cheaply split into subranges of any desired sizes, which makes it easy to divide work among parallel threads. The abstraction used by the streams library to perform this task is the *spliterator*, which is returned by the `spliterator` method on `Stream` and `Iterable`.

Another important factor that all of these data structures have in common is that they provide good-to-excellent *locality of reference* when processed sequentially: sequential element references are stored together in memory. The objects referred to by those references may not be close to one another in memory, which reduces locality-of-reference. Locality-of-reference turns out to be critically important for parallelizing bulk operations: without it, threads spend much of their time idle, waiting for data to be transferred from memory into the processor's cache. The data structures with the best locality of reference are primitive arrays because the data itself is stored contiguously in memory.

The nature of a stream pipeline's terminal operation also affects the effectiveness of parallel execution. If a significant amount of work is done in the terminal operation compared to the overall work of the pipeline and that operation is inherently sequential, then parallelizing the pipeline will have limited effectiveness. The best terminal operations for parallelism are *reductions*, where all of the elements emerging from the pipeline are combined using one of `Stream`'s `reduce` methods, or prepackaged reductions such as `min`, `max`, `count`, and `sum`. The *short-circuiting* operations `anyMatch`, `allMatch`, and `noneMatch` are also amenable to parallelism. The operations performed by `Stream`'s `collect` method, which are known as *mutable reductions*, are not good candidates for parallelism because the overhead of combining collections is costly.

If you write your own `Stream`, `Iterable`, or `Collection` implementation and you want decent parallel performance, you must override the `spliterator` method and test the parallel performance of the resulting streams extensively. Writing high-quality spliterators is difficult and beyond the scope of this book.

Not only can parallelizing a stream lead to poor performance, including liveness failures; it can lead to incorrect results and unpredictable behavior (*safety failures*). Safety failures may result from parallelizing a pipeline that uses mappers, filters, and other programmer-supplied function objects that fail to adhere to their specifications. The `Stream` specification places stringent requirements on these function objects. For example, the accumulator and combiner functions passed to `Stream`'s `reduce` operation must be associative, non-interfering, and stateless. If you violate these requirements (some of which are discussed in Item 46) but run your pipeline sequentially, it will likely yield correct results; if you parallelize it, it will likely fail, perhaps catastrophically.

Along these lines, it's worth noting that even if the parallelized Mersenne primes program had run to completion, it would not have printed the primes in the correct (ascending) order. To preserve the order displayed by the sequential version, you'd have to replace the `forEach` terminal operation with `forEachOrdered`, which is guaranteed to traverse parallel streams in *encounter order.*

Even assuming that you're using an efficiently splittable source stream, a parallelizable or cheap terminal operation, and non-interfering function objects, you won't get a good speedup from parallelization unless the pipeline is doing enough real work to offset the costs associated with parallelism. As a *very* rough estimate, the number of elements in the stream times the number of lines of code executed per element should be at least a hundred thousand [Lea14].

It's important to remember that parallelizing a stream is strictly a performance optimization. As is the case for any optimization, you must test the performance before and after the change to ensure that it is worth doing (Item 67). Ideally, you should perform the test in a realistic system setting. Normally, all parallel stream pipelines in a program run in a common fork-join pool. A single misbehaving pipeline can harm the performance of others in unrelated parts of the system.

If it sounds like the odds are stacked against you when parallelizing stream pipelines, it's because they are. An acquaintance who maintains a multimillion-line codebase that makes heavy use of streams found only a handful of places where parallel streams were effective. This does *not* mean that you should refrain from parallelizing streams. **Under the right circumstances, it *is* possible to achieve near-linear speedup in the number of processor cores simply by adding a `parallel` call to a stream pipeline.** Certain domains, such as machine learning and data processing, are particularly amenable to these speedups.

As a simple example of a stream pipeline where parallelism is effective, consider this function for computing $\pi(n)$, the number of primes less than or equal to n:

```
// Prime-counting stream pipeline - benefits from parallelization
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

On my machine, it takes 31 seconds to compute $\pi(10^8)$ using this function. Simply adding a `parallel()` call reduces the time to 9.2 seconds:

```
// Prime-counting stream pipeline - parallel version
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .parallel()
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

In other words, parallelizing the computation speeds it up by a factor of 3.7 on my quad-core machine. It's worth noting that this is *not* how you'd compute $\pi(n)$ for large values of n in practice. There are far more efficient algorithms, notably Lehmer's formula.

If you are going to parallelize a stream of random numbers, start with a `SplittableRandom` instance rather than a `ThreadLocalRandom` (or the essentially obsolete `Random`). `SplittableRandom` is designed for precisely this use, and has the potential for linear speedup. ThreadLocalRandom is designed for use by a single thread, and will adapt itself to function as a parallel stream source, but won't be as fast as `SplittableRandom`. `Random` synchronizes on every operation, so it will result in excessive, parallelism-killing contention.

In summary, do not even attempt to parallelize a stream pipeline unless you have good reason to believe that it will preserve the correctness of the computation and increase its speed. The cost of inappropriately parallelizing a stream can be a program failure or performance disaster. If you believe that parallelism may be justified, ensure that your code remains correct when run in parallel, and do careful performance measurements under realistic conditions. If your code remains correct and these experiments bear out your suspicion of increased performance, then and only then parallelize the stream in production code.

This page intentionally left blank