# A Thought on Refactoring Java Loops Using Java 8 Streams

Khandoker Rahad, Zejing Cao, and Yoonsik Cheon

TR #17-44
June 2017

**Keywords:** code refactoring, lambda expression, loop, pattern, stream, transformation rule, Java 8 Stream API.

**1998 CR Categories:** D.2.4 [*Software Engineering*] Software/Program Verification—formal methods; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement—restructuring, reverse engineering, and reengineering; D.3.3 [*Programming Languages*] Language Constructs and Features—control structures, data types and structures, patterns; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs—Assertions, logic of programs, specification techniques

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

# A Thought on Refactoring Java Loops Using Java 8 Streams

Khandoker Rahad, Zejing Cao, and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas 79968

***Abstract.*** *Java 8 has introduced a new abstraction called a* stream *to represent an immutable sequence of elements and to provide a variety of operations to be executed on the elements in series or in parallel. By processing a collection of data in a declarative way, it enables one to write more concise and clean code that can also leverage multi-core architectures without needing a single line of multithread code to be written. In this document, we describe our preliminary work on systematically refactoring loops with Java 8 streams to produce more concise and clean code. Our idea is to adapt existing work on analyzing loops and deriving their specifications written in a functional program verification style. Our extension is to define a set of transformation rules for loop patterns, one for each pattern, by decomposing the derived specification function into a series of stream operations.*

## 1. Introduction

Any non-trivial program contains loop control structures such as *while*, *for* and *do* statements, and code segments containing loops are oftentimes the most difficult part of a program to read and understand. It is also said that dealing with loops is the most difficult part of program analysis and formal verification [5]. Among the three loop statements (*for*, *while*, and *do* statements) found in C, C++, and Java, the *for* statement is the most frequently used [9].

Java 8 introduced several new language features including as lambda expressions and streams [6][8]. A *lambda expression* is a block of code with parameters that can be passed around so that it can be executed later. A *stream* is an immutable sequence of elements, providing a variety of so-called *higher-order operations* that take lambda expressions as arguments. The Java 8 Stream application programming interface (API) allows one to work with a sequence of elements possibly in parallel without worrying about how the elements are stored or accessed. One key benefit of using streams is internalization of iterations, called *internal iterations*. The code is completely unaware of the iteration logic in the background. This is contrary to the conventional way of iterating through an array or collection using an index or an iterator. As an example, consider the following code.

```
for (int i; i < players.length; i++) {
    players[i].setFleet(defaultFleet());
}
```

This conventional iteration is called an *external iteration*, and the iteration is clearly visible in the code, i.e., use of an index *i*. The Stream API provides methods like *forEach* to internalize iterations (see below), and there is no code written for the iteration logic.

```
Stream.of(players).forEach(p -> p.setFleet(defaultFleet())
```

The static method *Stream.of* creates a new stream from an array (*players*), and the *forEach* operation performs a specified action on each element (*p*) of a stream. The argument of the *forEach* operation is a lambda expression comprising of parameters, a lambda operator (->) and a function body. As shown, the use of streams along with lambda expressions produces concise code written at a higher level of abstraction.

In this document, we report our preliminary work on refactoring Java loops using Java 8 streams. The goal is to systematically refactor loop code to produce more concise and clean code at a higher level of abstraction. The key idea of our approach is to define a set of refactoring or transformation rules based on the patterns of loops. It is based on the observation that many loops exhibit certain common flavors or patterns, and similarly-structured loops have similarly-structured specifications, i.e., similar behaviors [1]. Thus, similarly-structured loops can be refactored to similarly-structured code written using Java 8 streams. We suggest to extend existing work on analyzing loops and deriving their specifications [1][2][3] to decompose the derived specification function into a series of stream operations.

The rest of this document is structured as follows. In Section 2 below, we provide a quick overview of Java 8 Stream API. Since the first step of any code refactoring is to study and know the code, we describe in Section 3 the technique to be used for analyzing loop code systematically. We also introduce a few representative patterns of loops. In Section 4, we first sketch our approach using an example and then discuss interesting research questions and issues. In Section 5, we conclude this document with a summary.

## 2. Java 8 Stream API

Since almost every program manipulates collections of data, the notion of collections is fundamental to many programming tasks. As thus, Java Software Development Kit (SDK) provides a set of interfaces and classes to represent different kinds of collections (see Figure *1*). These collections have different properties and characteristics, including the set of operations and their time and space complexities. For example, some collections allow duplicate elements and others do not. Some collections store their elements ordered while others do not.

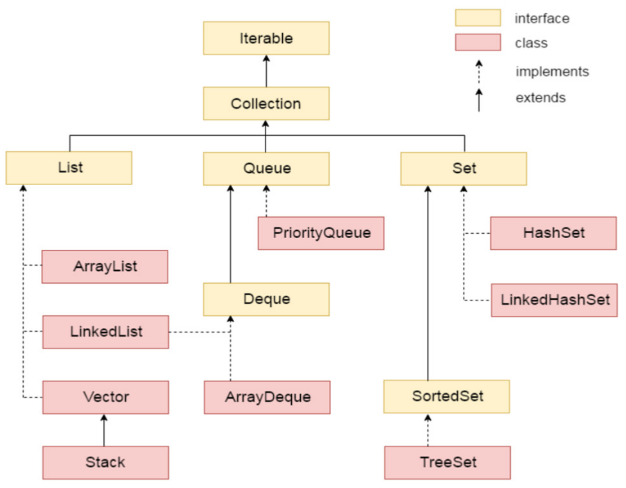

Figure 1. Java collections hierarchy

Java 8 introduced a new API to manipulate collections, called the Stream API [6][8]. A *stream* is an immutable sequence of elements and provides a variety of operations to be executed on the elements in series or in parallel. One key difference of streams from collections is that streams are centered around operations. Collections are in-memory data structures to hold elements within it, and each element of a collection is computed before it becomes a part of the collection. On the other hand, streams are fixed data structures that computes the elements on-demand basis. The Stream API allows one to work with a sequence of elements possibly in parallel without worrying about how the elements are stored or accessed. To perform a computation, stream operations are composed into a stream pipeline (see Figure 2). A stream pipeline consists of a source, zero or more intermediate operations and a terminal operation. Streams are most often lazy in that computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed. A few key features of the Stream API include:

- *Functional-style operations*: A variety of higher-order operations are provided, including *filter*, *map*, and *reduce* (also called *fold* in some functional languages).
- *Lazy construction*: A stream is constructed lazily in that its elements are computed when a user demands it. This is contrary to a collection whose elements are constructed eagerly in that the elements are computed before they become parts of the collection.
- *Concurrency*: Many parallel operations are provided to process the elements contained in a stream, while completely abstracting out the low level multithreading logic.
- *Pipeline:* The API is based on the idea of converting a collection to a stream, processing the elements possibly in parallel, and then gathering the results into a collection. The elements are processed by pipelining stream operations, zero or more so-called *intermediate* operations like *map* followed by a *termination* operation like *reduce* [8].

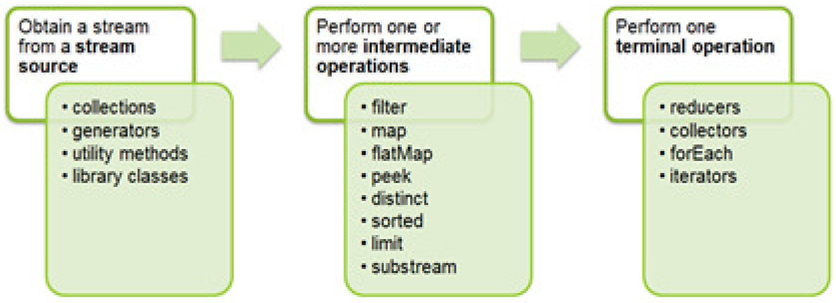

Figure 2. Using streams

A typical use of streams involves three steps: (a) creating a stream, (b) forming a pipeline, and (c) closing the pipeline to return a result (see Figure 2). The *java.util.Stream* interface defines many operations grouped into two categories.

- *Intermediate operations*: Operations that can be connected together to form a pipeline.
- *Terminal operations:* Operations that close the pipeline and return a result.

Some of these operations are summarized in Table 1. The code snippet below shows how to use a stream. It calculates squares of even numbers contained in a list by first creating a stream from a list, pipelining two intermediate operations (*filter* and *map*), and then applying a terminal operation (*collect*). The *Collectors.toList*() returns a collector that accumulates the elements into a new list.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> evenSquares =  numbers.stream()
        .filter(n -> n % 2 == 0)
        .map(n -> n * n)
        .collect(Collectors.toList());
```

Table 1. Common stream operations

| Operation | Description |
|---|---|
| boolean allMatch(Predicate<? super T> predicate) | Does every element satisfy the provided predicate? |
| boolean anyMatch(Predicate<? super T> predicate) | Does any element satisfy the provided predicate? |
| <R,A> R collect(Collector<? super T,A,R> collector) | Performs a mutable reduction on the elements using the provided collector. |
| long Stream<T> count() | Returns the number of elements. |
| Stream<T> distinct() | Returns a stream consisting of all distinct elements. |
| static <T> Stream<T> empty () | Returns an empty sequential stream. |
| Stream<T> filter(Predicate<? super T> predicate) | Returns a stream consisting of the elements that satisfy the given predicate. |
| void forEach(Consumer<? super T> action) | Performs a provided action on each element. |
| static <T> Stream<T> iterate(T seed,  UnaryOperator<T> f) | Produces a stream consisting of seed, f(seed), f(f(seed)), etc. |
| <R> Stream<R> map(Function<? super T? extends R> mapper) | Returns a stream consisting of the results of applying the given mapper to the elements. |
| Optional<T> max(Comparator<? super T> comparator) | Returns the maximum element according to the provided comparator. |
| Optional<T> min(Comparator<? super T> comparator) | Returns the minimum element according to the provided comparator. |
| T reduce(BinaryOperator<T> accumulator) | Performs a reduction on the elements using the accumulator. |

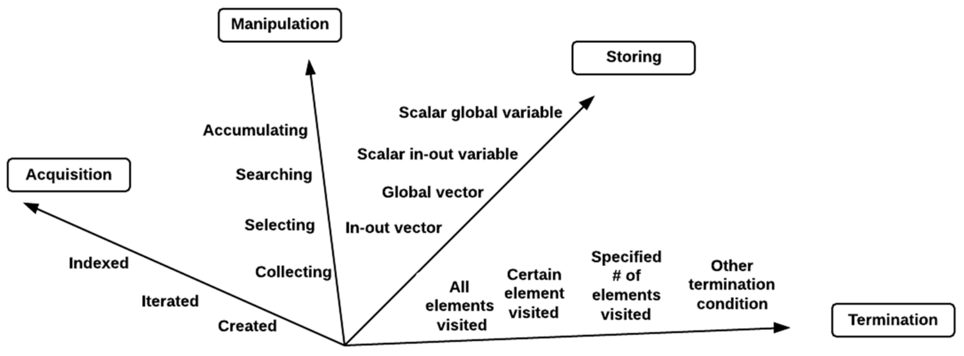

Figure 3. Dimensions of loop analysis

## 3. Loop Analysis and Patterns

The first step of code refactoring is of course to analyze the structure and behavior of the loop code under consideration. We use the loop analysis framework proposed by Barua and Cheon [1][3], and the framework is summarized in this section. It is said that  the most common use of loops is to iterate over a certain sequence of values and manipulate the iterated values. And thus, a loop has a chain of steps that are performed and then repeated. It was suggested to study and identify four different types of steps or actions in the chain (see Figure 3):

a) How the next value is obtained?
b) What manipulation is performed on the value obtained?
c) Where and how is the (manipulation) result stored?
d) When does the loop terminate?

As an example, consider the following code snippet containing a *while* loop.

```
1  while (i < a.length) {
2    if (a[i] > 0)
3        r = r + 1;
4    i++;
5  }
```

The sequence iterated over by the loop is the elements of the array *a* starting from index *i* to the end in order, i.e., *a*[*i..a.length - 1*], and an analysis of the loop shows the following.

a) Acquiring values: an index (*i*) is used to access elements of *a* (*a*[*i*] in line 2) and each element is accessed in order (*i*++ in line 4).
b) Manipulating values: if the current value is positive, compute *r* + 1 (lines 2-3).
c) Storing results: the manipulation result is stored in a scalar variable *r* (line 3). Assuming that *i* is an *incidental variable* used only for iterating over the sequence, there is only one non-local variable that may be updated or changed.
d) Determining termination: the loop terminates when the last element is used (line 1), i.e., when it completes iteration over all elements of *a* starting from index *i*.

It is said that there is a wide range of possibilities along the four analysis dimensions, including several most commonly-used ones as described in Figure 3. The acquisition dimension tells how the loop acquires the next value of the sequence being iterated over. The sequence is frequently stored explicitly in such structures as arrays, strings, collections, streams, and files, and its elements are accessed by using indices or various forms of iterators. It is also possible to create the elements on-the-fly on a need basis instead of retrieving stored ones. The manipulation dimension determines the functionality of a loop by telling how the acquired values are manipulated or what operations are performed on them. It is often the most important analysis dimension, for it represents the purpose of a loop. As expected, there are numerous manipulations possible, including several common types such as accumulating, searching, counting, selecting, and collecting. The storing dimension tells where and how the results are stored. There are also a variety of possibilities here, e.g., updating the input sequence and storing to output variables distinct from that of the input sequence. For accumulation and searching, the results are stored in scalar variables while for selecting and collecting, they are stored in vector or collection variables. The termination dimension specifies the termination condition of a loop --- a condition that stops the looping. It is the opposite of the test condition that allows the loop to continue iterating. A loop termination condition can differ in many ways, e.g., when all elements are accessed, when a certain element is accessed, and when a certain number of iterations are completed. The conditions may be written in terms of indices, iterators, values of the input sequence, and others. The four dimensions allow one to analyze a loop in a modular, compositional fashion by examining each dimension separately and composing the results. For example, the above while loop is a composition of an index-based sequential acquisition, a counting manipulation, a scalar variable storage or update, and termination upon accessing all elements. In Section 4, we will show how we use the analysis information in refactoring a loop.

Barua and Cheon noticed that many loops exhibit certain common flavors or patterns, and they categorized and documented the common flavors or usage patterns of loop control structures as reusable specification patterns [2][3]. The key idea of their pattern specifications is to promote the manipulation of individual values to the whole sequence iterated over by a loop. Since we will be using these patterns in our refactoring approach, some of them are summarized below.

- *Accumulating*: this pattern documents loops that iterate over a collection of values and combine some of the values into a single value, e.g., a loop that multiplies all positive values contained in an array. Figure 4 summarizes this this pattern. The behavior of a loop is specified in terms of that of the loop body and is written formally using the notation of *functional program verification* called a *concurrent assignment* of the form [$x_1$,$x_2$,…, $x_n$ := $e_1$, $e_2$, …, $e_n$] stating that each variable $x_i$'s new value in the post-state is an expression $e_i$ evaluated concurrently in the pre-state [4][7]. Refer to [2] for a detailed description and explanation of this pattern and its specification.
- *Searching*: this pattern documents those loops that search for a particular element in a collection. The result of such a loop is typically the element found, but other results are possible, e.g., the position or index of the element found and a flag indicating whether an element is found or not.
- *Selecting*: this pattern documents those loops that select some elements of a collection and store the selected elements to the same or a different collection.

**Name**: Accumulating
**Purpose:** Combine certain elements of a collection
**Description:** A loop combines certain elements of a collection into a single value by applying various binary (accumulating) operators such as addition, multiplication, and concatenation. The type of the accumulate value is often the same as that of the elements of the collection being accumulated.
**Structure:**

```
[r, i := $\overline{\oplus}$(r, s@i..), anything]
while (C) {
  [r, i := P(s@i) ? (r $\oplus$ s@i) : r, E(i)]
}
where
  s: collection whose elements are accumulated
  r: result variable accumulating elements of s
  i: abstract iterator to access the element of a collection
  s@i: i-th element of a collection s
  $\oplus$: binary accumulation operator such as +, -, and *
  $\overline{\oplus}$: promotion of $\oplus$ to a collection
  P(x): predicate written in terms of x
  E(i): expression written in terms of an abstract iterator i
```

**Applicability**: Arrays, strings, collections, etc.
**Variations:** Large number of variations on selection, accumulators, manipulation, acquisition, and storage.
**Related patterns**: Unconditional Accumulating, …
**Example:**

```
[r, i := r + $\sum_{j=i..a.length-1}$ a[i], anything]
while (i < a.length) {
  [r, i := r + a[i], i + 1]
    r = r + a[i];
    i++;
}
```

Figure 4. Accumulating pattern

# 4. Refactoring to Stream Code

## 4.1 Overview

As expected, the refactored code will have the general form of $s.o_1().o_2().,\ldots,o_n().t()$, where $s$ is a source stream, $o_i$'s are a series of intermediate stream operations, and $t$ is a terminal stream operation. Thus, the main refactoring tasks include creating a source stream $s$, identifying a set of intermediate and terminal stream operations ($o_i$'s and $t$), and chaining the stream operations into a series to connect them and return the result (see Figure 1).

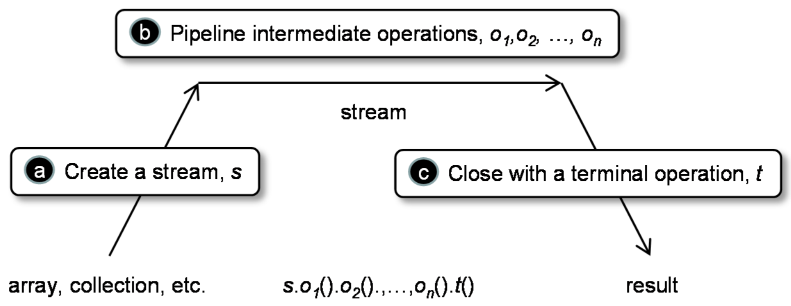
b Pipeline intermediate operations, $o_1, o_2, \ldots, o_n$
stream
a Create a stream, $s$
c Close with a terminal operation, $t$
array, collection, etc.
$s.o_1().o_2().,\ldots,o_n().t()$
result

Figure 5. Refactored code and refactoring tasks

The key idea of our approach is to define a set of transformation rules for loop patterns, one for each pattern, and apply the transformation rule of the pattern matched for a loop. For this, we use Barua and Cheon's loop analysis framework as well as loop patterns, described in the previous section. Their framework not only allows us to analyze loops systematically but also identify loop patterns easily. The loop analysis facilitates refactoring tasks described in Figure 1. For example, the source stream is created based on the acquisition and the termination of the loop. The intermediate operations are identified and pipelined based on the manipulation of the loop, and the optional terminal operation is determined according to the storing dimension of the analysis. We propose the following process for refactoring loop code to stream code.

(1) Analyze loop code along the four dimensions: acquisition, manipulation, storing, and termination.
(2) Identify a loop pattern by matching the loop to patterns in the pattern catalog [2].
(3) Transform the code by applying the transformation rule of the matched pattern
    (a) Create a source stream
    (b) Determine and pipeline intermediate operations
    (c) Determine an optional terminal operation

## 4.2 Example

In this subsection, we sketch out our approach by considering a simple example loop. A study indicates that 60% of loops written in C traverse arrays in some fashion, 45.2% for non-string arrays and 14.3% for string arrays and linked lists account for 13.0% [9]. Thus, we consider a loop that iterates over an array. The example loop code is shown below, and it is the same as the one shown in the previous section except that the loop control variable $i$ is initialized to 0.

```
1 i = 0;
2 while (i < a.length) {
3   if (a[i] > 0)
4       r = r + 1;
5   i++;
6 }
```

Recall that the first two steps are to analyze the loop code and match it to a loop pattern. We already performed an analysis of this loop in the previous section, and below are summarized the results again.

- Acquisition: an index (*i*) is used to access elements of *a* (*a*[*i*] in line 3) and each element is accessed in order starting from index 0 to *a.length* – 1 (*i*++ in line 5).
- Manipulation: if the current value is positive, compute *r* + 1 (lines 3-4).
- Storing: the manipulation result is stored in a scalar variable *r* (line 4). Assuming that *i* is an *incidental variable* used only for iterating over the sequence, there is only one non-local variable that is updated or changed.
- Termination: the loop terminates when the last element is used (line 2), i.e., when it completes iteration over all elements of *a*.

The loop matches the *Accumulating* pattern [2]. This pattern documents loops that iterate over a collection of values and combine some of the values to a single value, and it is rigorously defined in terms of a predicate for selecting elements, a binary operator for accumulating selected values, and others (refer to Section 3 and [2]). It is said that the Accumulating pattern is one of most commonly-used pattern.

The next step is to transform code by applying the transformation rule of the matched pattern. We believe that we can define transformations rules for all the patterns cataloged in [2]. Below we describe informally one possible transformation for our example.

(a) *Create a source stream*. To create a source stream, we need to identify the sequence of values iterated over by the loop. We use the acquisition and the termination analyses performed above to identify the input array *a* and the iteration of all the elements of *a* from index 0 to the end in order. Thus, the source stream can be created with an expression: *IntStream.of*(*a*). The *IntStream* interface represents a sequence of primitive *int*-valued elements, and the static *of*() method creates a new *IntStream* object from a given array.

(b) *Determine and pipeline intermediate operations*. The manipulation analysis says that *r* is incremented by 1 for each positive value iterated over. This is the same as converting each positive value to 1 and adding them together. The Accumulating pattern also document the selection predicate *P* and the accumulation operator ⊕ (see Section 3). The selection predicate *P* is instantiated to *a*[*i*] > 0 and can be easily transformed to the *filter* stream operation: *filter*(*v* -> *v* > *0*). And the conversion of positive values to 1's can be achieved by using the *map* stream operation: *map*(*e* -> 1). These two operations can be combined to form a pipelined expression: *filter*(*v* -> *v* > *0*).*map*(*v* -> 1).

(c) *Determine an optional terminal operation*. The result is stored in a scalar variable *r* by adding 1 for each positive element that is iterated over. Since all positive elements are already mapped to 1 in the above step, the terminal operation can be written with the *reduce* stream operator: *reduce*(*0*, (*x*, *y*) -> *x* + y). Note that the accumulation operator ⊕ is instantiated to + and the *reduce* operation is equivalent to $\overline{\oplus}$, its promotion to the whole sequence. Since the last stream of the pipeline will be a sequence of 1's, the *reduce* operation can be simplified to *count*() or *sum*().

If we combine all the sub-expressions produced above, we get the following stream-based expression.

```
r = r + IntStream.of(a).filter(v -> v > 0).map(v -> 1).reduce(0, (x, y) -> x + y);
```

The expression can be simplified to: *r* = *r* + *IntStream*.*of*(*a*).*filter*(*v* -> *v* > 0).*count*().

## 4.3 Formulating Transformation Rules

We believe that we can formulate the refactoring heuristics like the one in the previous subsection into transformation rules. It will be also possible to tune the level of formality in defining the transformation rules, say from informal to rigorous and formal. Remember that the loop patterns are specified formally in the *functional* program verification style (see Section 3). This is also encouraging in that our refactoring approach is to convert imperative code into functional-style one. However, there are interesting research questions and issues.

As mentioned shown in the description of the loop analysis framework (see Section 3), there are many different ways that a loop can access the values that it iterates over: source types (arrays, collections, strings, files, network connections, on-the-fly generation, etc.), access mechanisms (indices, iterators, iterables, references, etc.), and orders and numbers of elements (every element, every *n*-th element, condition-based, etc.). Obviously, the creation of a source stream is also influenced by the termination condition of a loop and the manipulation of the values in the loop body as well. If a loop performs a certain manipulation, say *M*, it would be advantageous to create a stream that provides *M* or closely-related operations as built-in stream operations. Java 8 provides several different types of streams, including such interfaces as *Stream*, *IntStream*, *LongStream*, and *DoubleStream*. Different stream types provide different sets of stream manipulation and creation operations. For example, an *IntStream* object can be created by specifying a range of values (start and end), and its values can be summed using the *sum*() operation. It would be an interesting research problem to come up with transformation rules for converting collections and arrays to appropriate streams (see Figure 6). The rules should be universal or general enough to accommodate various ways of accessing values by loops and at the same time be specific or concrete enough to be applicable to actual code. To facilitate the transformation, it may be beneficial to introduce custom streams and other supporting library classes and methods.

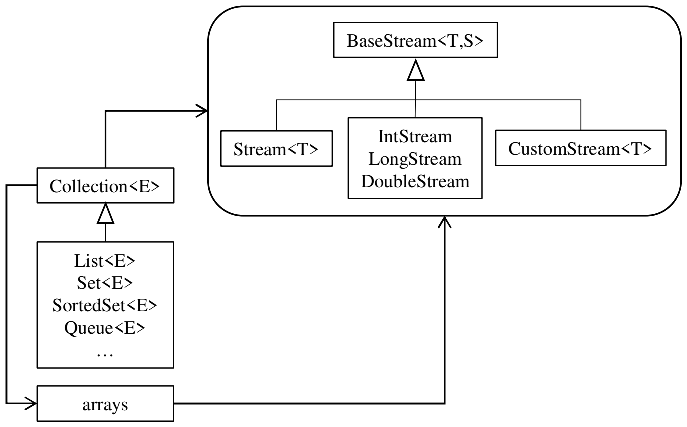

Figure 6. Creating streams from collections and arrays

The major part of the transformation rules will be concerned with identifying and pipelining intermediate operations and then closing the pipeline with a terminal operation. Our idea is to define the transformation rules based on the behavior of the loops. Such rules will be applicable to many different loops of the same behavior regardless their code structures and coding specifics. However, it will require rigorous, systematic analyses of the loops. Our approach is to adapt the technique of Barua and Cheon [1] [3]. The technique is well suited for our purpose because it uses a functional program specification and verification technique where a program is viewed and specified as a mathematical function from one program state to another. Our adaptation is to decompose the behavior of the loop --- so-called an *intended function* in the functional verification technique [4][7] --- into a series of stream operations. Thus, the problem can be formulated as: given loop code with an intended function $f$ that accesses and manipulates a collection of values, $c$, find a series of stream operations, $o_1, o_2, \ldots, o_n$, such that $f \equiv s.o_1().o_2,(), \ldots, o_n()$, where $s$ is a stream consisting of all the elements of $c$ iterated over by the loop. The last operation $o_n$ may be a terminal operation that closes the pipeline and calculates the final result. We will need to analyze the loop body and its intended function to find both the manipulation of individual values and their composition. These two analyses can be done separately in a modular fashion as suggested by Barua and Cheon [1] [3], though the main focus would be on the manipulation of individual values. In general, a manipulation $M$ of a collection can be decomposed into a series of stream operations consisting of *filter*, *map*, and *reduce* (also called *fold*). However, one complication would be data dependence, e.g., the manipulation of $i$-th value may depend on those of preceding values. This will require an analysis of data dependency among individual values, and to define intermediate operations such a way to calculate dependency conditions or constraints and pass them through the pipeline (i.e., intermediate streams).

## 5. Summary

We described our thought on systematically refactoring Java loops using Java 8 streams to produce more concise and clean code written at a higher level of abstraction. The code refactoring is essentially defining an expression $s.o_1().o_2,(), \ldots, o_n()$, where $s$ is a stream consisting of all the values iterated over by the loop, and $o_i$'s are stream operations. Our idea is to match a loop to a pattern that documents the behavior of the loop in the functional program verification style and then decompose the documented behavior (written as a function) into a sequence of stream operations. The latter can be facilitated by defining a transformation rule for a loop pattern.

## References

[1] Aditi Barua and Yoonsik Cheon, Finding Specifications of While Statements Using Patterns, *New Trends in Networking, Computing, E-learning, Systems Sciences, and Engineering, Lecture Notes in Electrical Engineering,* Volume 312, November 2014, pages 5381-588, Springer. DOI: 10.1007/978-3-319-06764-3_75

[2] Aditi Barua and Yoonsik Cheon, *A Catalog of While Loop Specification Patterns,* Technical Report 14-65, Department of Computer Science, University of Texas at El Paso, El Paso, TX, September 2014.

[3] Aditi Barua and Yoonsik Cheon, *A Systematic Derivation of Loop Specifications Using Patterns,* Technical Report 15-90, Department of Computer Science, University of Texas at El Paso, El Paso, TX, December 2015.

[4] Yoonsik Cheon, Cesar Yeep, and Melisa Vela. The CleanJava language for functional program verification. *International Journal of Software Engineering*, 5(1):47–68, January 2012.

[5] Eric Larson. Program analysis too loopy? set the loops aside. *IET Software*, 7(3):131–149, June 2013. DOI: 10.1049/iet-sen.2012.0048.

[6] Jame Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley, *The Java Language Specification, Java SE 8 Edition*, February, 2015, Available from: https://docs.oracle.com/javase/specs/jls/se8/html/index.html.

[7] Allan M. Stavely. *Toward Zero Defect Programming*. Addison-Wesley, 1999.

[8] Richard Warburton, *Java 8 Lambdas, Functional Programming for the Masses*, O'Reilly, 2014.

[9] Xiaoyan Zhu, E. James Whitehead, Caitlin Sadowski, and Qinbao Song. An analysis of programming language statement frequency in C, C++, and Java source code. *Software: Practice and Experience*, 45(11):1479–1495, 2015. DOI: 10.1002/spe.2298.