Impacts of Java Language Features On the Memory Performance of Android Apps

Adriana Escobar Del La Torre and Yoonsik Cheon

TR #17-84
September 2017

Keywords: garbage collection, lambda expression, memory allocation, mobile app, performance, Android, Java 8 Stream API.

1998 CR Categories: D.2.3 [*Software Engineering*] Coding Tools and Techniques—object-oriented programming; D.3.3 [*Software Engineering*] Language Constructs and Features—control structures, classes and objects, data types and structures, dynamic memory management, patterns; D.4 [*Operating Systems*] Storage Management—garbage collection

To be presented at the *28th Annual HENAAC Conference*, Pasadena, CA, October 18-22, 2017.

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

Impacts of Java Language Features on the Memory Performances of Android Apps

Adriana Escobar De La Torre and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas 79968

Abstract. *Android apps are written in Java, but unlike Java applications they are resource-constrained in storage capacity and battery lifetime. In this document, we perform an experiment to measure quantitatively the impact of Java language and standard API features on the memory efficiency of Android apps. We focus on garbage collection because it is a critical process for performance affecting user experience. We learned that even Java language constructs and standard application programming interfaces (APIs) may be a source of a performance problem causing a significant memory overhead for Android apps. Any critical section of code needs to be scrutinized on the use of these Java features.*

1. Introduction

Android is one of the most popular mobile platforms paving the way for the development of a flood of mobile apps. Android devices are resource-constrained in storage capacity and battery lifetime. Performance is always a problem for anyone developing Android apps [12] [15]. Memory, for example, is a lot more valuable on Android than on other operating systems. An application launched on Microsoft Windows, for example, may stay running indefinitely. It is different on Android. Android has a memory conservation mechanism known as Low Memory Killer (LMK) [14]. When too much memory is used, LMK will start killing background and inactive processes that consume large amounts of memory. Thus, Android programmers should build apps with memory conservation in mind.

Android apps are written in Java though there are some minor differences between the Java application programming interface (API) and the Android API. Because of this, there is a misconception that the best Java programming and coding practices -- tips, idioms, styles, patterns, and recipes -- are equally applicable to Android programming [2]. In fact, some are anti-patterns or code smells [10] that Android programmers should avoid. They have devastating effects on the memory performances of Android apps, causing frequent garbage collection and thus making the apps unusable. It is also reported that frequent garbage collection may affect the battery life of a device by overheating it [3].

In this document, we perform an experiment to ask a more fundamental question: what are the impacts of the Java language and standard API features on the memory efficiency of Android apps? We consider, among others, the following Java features: lambda expressions (Java 8), Stream API (Java 8), for-each statements (Java 5) and iterators. These are features that in general produce more succinct and cleaner code [11]. Java 8 is a big step for the Java language [8], and features like the lambda notation are most anticipated and long-awaited features for Java and Android. We design our experiment in such a way to measure quantitatively the garbage collection frequencies as well as the amount of additional memory allocated due to the above mentioned Java features.

Our key finding is that all the above Java features (except for for-each statements on arrays) allocate hidden objects. The objects are hidden in the sense that they are caused by the way the features are translated by the compiler or evaluated at runtime, and oftentimes programmers may not be aware of them. In general, this may not be such a big deal regarding the memory consumption of an app; however, if the code containing the features are executed repeatedly and continuously, there can be significant memory

overheads, e.g., a dozen garbage collection events per second. Our experiment also shows that the underlying assumption of object-oriented programming (OOP) -- objects are cheap and do not take lots of resources such as memory – does not hold on Android. The idea of OOP is to let many small objects solve a task together, each object focusing on a small aspect of the task. Android programmers, however, need to scrutinize any critical section of code for inefficient use of memory. It is important to analyze and know the memory impacts of the so-called best Java coding practices, as they in a certain situation may have devastating effects on the memory performance of an app [2]. One unpleasant finding from our experiment is that one needs to pay close attention to hidden objects caused by not only one's own code but also the built-in language constructs like the for-each statement and the Java 8 lambda notation.

2. Android Java: Language and API features

Java 8 is a big step for the Java language [8]. It makes it easy for programmers to create new applications and clean up existing code by writing cleaner code. The two most significant features of Java 8 are lambda expressions (see Section 15.27 of [8]) and the Stream application programming interface (API). Indeed the support for the lambda notation is one of the most anticipated and long-awaited features for Java and Android. At the time of writing, however, Android supports a subset of Java 8 language features that vary by platform version[1]. In this study, we consider these two Java 8 features along with two other features from earlier versions of Java as listed below.

- Iterators
- For-each statement
- Lambda expressions
- Stream API

Iterators: Java provides an interface named *Iterator* to access the elements of various collections in a uniform way. The interface implements the Iterator design pattern [4] and defines methods like *hasNext*() and *next*() to iterate over elements of a container or aggregate object. As the interface provides a way to access the elements without exposing the underlying representation, its use is a recommended coding practice. As expected, all Java collection classes implements the interface.

For-each statement: This statement also known as the enhanced *for* statement is introduced in Java 5. As shown below, it provides an even simpler way to iterate through the elements of a collection.

```
int[] values = …;
for (int v: values) {
    sum += v;
}
```

The for-each statement works for an array and any class that implements the *Interable* interface. The *Iterable* interface defines an *iterator*() method that returns an *Iterator* object. All Java collection classes implements the *Iterable* interface. As there is no need to introduce and manipulate an index or loop variable to access the elements of a collection, the for-each statement produces cleaner code.

Lambda expressions: A *lambda expression* introduced in Java 8 is a block of code with parameters that can be passed around so that it can be executed later [8] [13]. It is one of the most anticipated features of Java, and it allows one to pass to a method not only data but also a behavior, thus enabling to dramatically raise the level of abstraction. As shown below, the lambda notation can be used to omit the boilerplate code

[1]Currently the Jack tool chain is needed to use Java 8, which compiles Java source code into Android bytecode. In the preview version of Android Studio 3, Java 8 language features are built into the default tool chain.

for implementing an interface like the *ActionListener* interface — i.e., defining an anonymous or named class and creating an instance of it.

```
JButton playButton = new JButton("Play");
playButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
      startNewGame();
  }
});

// rewritten in lambda notation
playButton.addActionListener(e -> startNewGame());
```

A lambda expression comprises of parameters, a lambda operator (->) and a function body. It can be used in a place where an object of a *functional interface* is required. A functional interface is an interface with a single method; for example, the *ActionListener* interface in the above code is a functional interface.

Streams: Java 8 introduced the Stream application programming interface (API) [13]. A *stream* is an immutable sequence of elements and provides a variety of operations to be executed on the elements possibly in parallel. 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. The code snippet shown below calculates the 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 of a stream in 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());
```

One key difference between streams and collections is that streams are centered on operations while collections are in-memory data structures to hold elements within them. That is, 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. 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. 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. As shown in the above sample code, the use of streams along with lambda expressions produces functional-style code of a higher level of abstraction.

3. Experiment

The purpose of our experiment is to study the impacts of Java language and API features on the memory performance of Android apps. Our study is focused on garbage collection (GC) because it is a critical process for performance that can affect user experience of an app. GC can impair the performance of an app, resulting in choppy display and poor UI responsiveness. Android Studio provides a tool called *Memory Monitor* to visualize the real-time information about the memory usage of an app, e.g., graphs showing available and allocated memory over time as well as garbage collection events over time [5]. Since it can display the patterns of GC events graphically, it is a good tool to profile and optimize memory use of an app.

However, we need to have more accurate memory usage information to find out the memory overheads of the Java features under investigation. We write sample code of the features that also includes small probe to collect information about GC events as well as the memory use.[2] We write the simplest sample code to prevent any overheads not attributed to the features under investigation. Table 1 shows our sample code along with equivalent code written without using the features. We will compare the memory usage of the two code to find out the memory overheads caused by the features.

Table 1. Sample code of Java features under investigation

Feature	Code	Refactored
Iterator	**int** sum; List<Integer> values = …;	

	Iterable<Integer> it = values.iterator();
	while (it.hasNext()) {
	  sum += it.next();
	}
		for (**int** i = 0; i < values.size()) {
		  sum += values.get(i);
		}

For-each	**int** sum; List<Integer> values = Aarrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);	

	for (**int** x: values) {
	  sum += sum;
	}
		for (**int** i = 0; i < values.size()) {
		  sum += values.get(i);
		}

Lambda	**int** sum;
	void helper(Runnable action) {
	  action.run();
	}
	Runnable action = () -> sum++;	

	helper(() -> sum++);	helper(action);
Stream	**int** sum; **int**[] values = …;	

	sum = Arrays.stream(values)
	  .filter(x -> x > 0)
	  .map(x -> x + 1)
	  .sum();
		for (**int** x: values) {
		  if (x > 0) {
		  sum += (x + 1);
		  }
		}

For each sample code, we measure several different factors that indicate memory usage and performance of the code, including:

- Interval between two consecutive GC events
- Frequency of GC
- Allocated heap size at the time of a GC event
- (Hidden) objects created due to the language features under investigation
- Number of times the code is executed to trigger a GC event

[2]Some of the information can also be obtained using the Android Debug Bridge (adb) command-line tool. This tool can measure GC performance of apps, i.e., GC timing dump and systrace [7].

The first two can be calculated by measuring the number of GC events for a certain period of time. The third can be measured by finding out the allocated heap size when a GC is initiated, and the last can be measured in a similar fashion. The Memory Monitor tool of the Android Studio can be used to find out the objects that are allocated due to the features under investigation, either by dumping the Java heap or tracking memory allocation time [5].

Most of the measurements are done by our probe code mentioned previously. The key idea for writing the probe is to create a garbage that will be collected when garbage collection is initiated – i.e., when a GC event happens. As shown below, we create such a garbage by using a *weak reference*, a reference that does not protect the referenced object from being collected by a garbage collector.

```
private WeakReference<GarbageCollectionWatcher> gcWatcher
      = new WeakReference<>(new GarbageCollectionWatcher());
private class GarbageCollectionWatcher {
    protected void finalize() throws Throwable {
        // collect memory usage information here …
        gcWatcher = new WeakReference<>(new GarbageCollectionWatcher());
    }
}
```

The *gcWatcher* field is a weak reference. Its referent, an instance of the GarbageCollectionWatcher class, is a garbage that will be collected when garbage collection begins. We use the *finalize*() method to collect various information about the memory usage. The method also re-initialize the *gcWatcher* field to detect the next garbage collection. The *finalize*() method is called by the garbage collector.

To perform the experiment, we wrote an Android app that runs our sample code. The app was written with Android Studio 2.3.3 configured to use the Jack tool chain in order to use Java 8 language features such as the lambda notation and the Stream API. The app runs sample code repeatedly in a loop until a user click the stop button, at which time it displays the collected memory usage information (see a screenshot in the left). A user can select either the code written with or without using the feature experiment. Since the Java 8 Stream API is available only on Android API level 24 or higher, the app works only on Android version 7.0 (Nougat) or above. This also means that our experiment is done on the Android Runtime (ART), not its predecessor Dalvik virtual machine. The ART is the default Android runtime for Android 5.0 and beyond. It is said that ART provides an improved GC mechanism over Dalvik by having several different garbage collection plans to run different garbage collectors [5]. The default GC plan is the concurrent mark sweep (CMS), which uses mostly sticky CMS and partial CMS. The sticky CMS scans only the portion of the heap that was modified since the last GC and thus can reclaim only the objects allocated since the last GC.

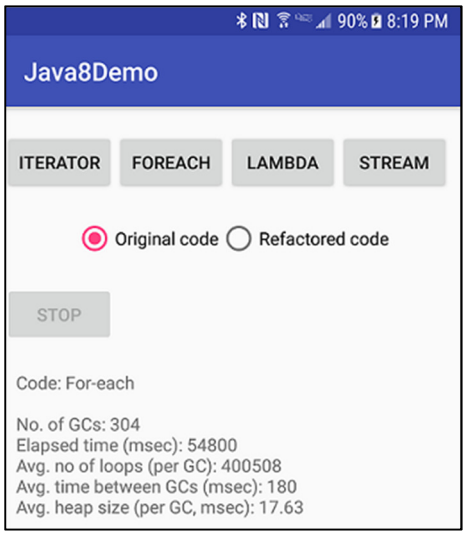

Figure 1. Experiment app

4. Results

We ran our experiment app on a Samsung Galaxy S7 smartphone with Qualcomm Snapdragon 820 2.1Ghz Quad-Core CPU and 4GB RAM running Android version 7.0. Table 2 show the results of running our sample code with the features under investigation; there was no additional memory allocation for the refactored code -- code without the features -- and thus no garbage collection either. The fourth column (*Avg. # loops*) shows the number of times the code under measurement is executed between two GC events; remember that our app executes the code repeatedly in an infinite loop until the user clicks the stop button. The sixth column (*Avg. heap size*) shows the average amount of allocated memory upon GC events. For all the features under investigation, a lot of GC events occur over and over again in a short period of time, i.e., 4 - 12 times per second. We will look into this table closely in the next section.

Table 2. Results of running the experiment app

	# GC	Execution time (msec)	Avg. # loops (/gc)	Avg. GC cycle (msec)	Avg. heap size (mb)	GC freq (gc/sec)
Iterator	377	60328	435113	160.02	17.28	6.25
For-each	251	43740	408477	174.26	18.14	5.74
Lambda	648	72299	2106384	111.57	27.41	8.96
Stream	691	55374	41265	80.14	23.46	12.48

We also looked at the memory usage of our app using the Memory Monitor tool of the Android Studio (see Figure 2). The amount of allocated memory changes quite frequently and fluctuates for all features due to GC, and there are significant differences in the peak amounts of allocated memory, i.e., about 16 MB for iterators and for-each statements and about 32 MB for lambdas and streams. When no feature is used (i.e., refactored code), however, the graph is flat, which is the ideal scenario from the performance perspective.

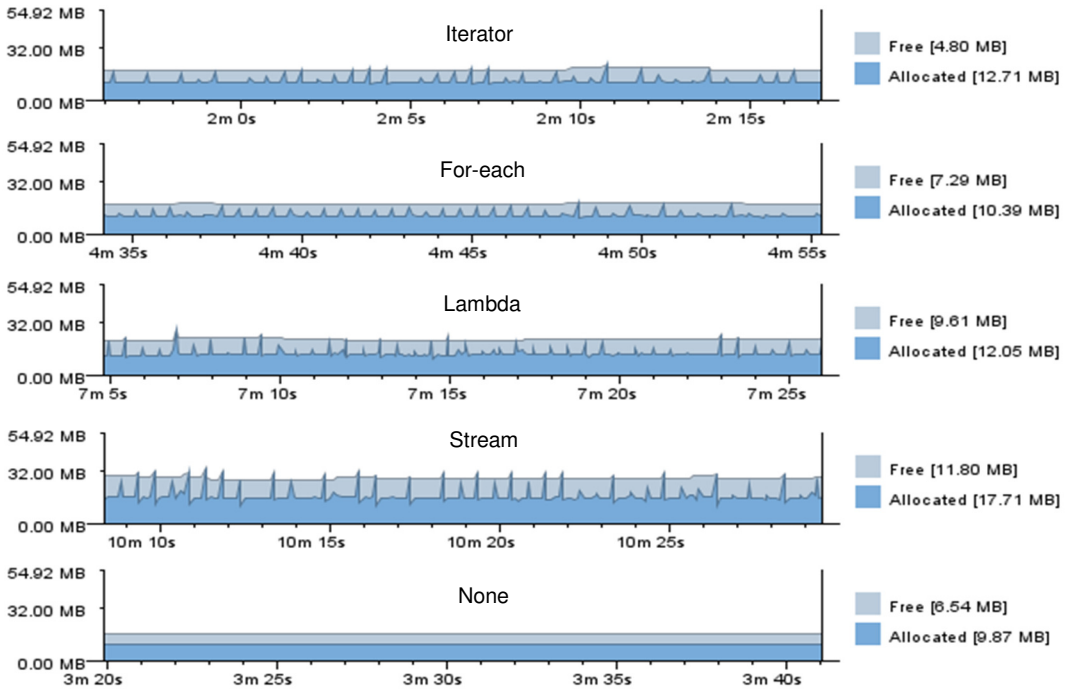

Figure 2. Patterns of garbage collection events

We also performed our experiment on emulators of different configurations. The main motivation was to limit the heap size for the app. It is impossible to do so on a real device because the heap limit of an app is determined by the device itself, not the app. On Android virtual device (AVD), however, one can configure its heap size when it is created. The findings from this experiment will be described in the following section.

5. Observations and Analyses

The first observation we made is that all the Java features under investigation have overheads regarding memory allocations (see Table 3). They require additional memory to be allocated in the heap storage. For all the features, the amount of allocated memory does not change -- meaning no additional memory allocation – when they are not used. With the features, however, the amount of allocated memory fluctuates due to GC events. The table shows the average amount of allocated memory upon GC events. Additional memory needed are in the range of 11-21 MB, giving overheads from 134% up to 311%. It should be noted, however, that this is a worst case analysis in that we run the code repeatedly and continuously in a loop (see Section 6 for more on this).

Table 3. Memory overheads

Feature	Allocated memory (MB)		Difference	Overhead (%)
	w/o feature*	with feature**		
Iterator	6.54	17.28	10.74	164.22
For-each	6.54	18.14	11.06	177.37
Lambda	6.67	27.41	20.74	310.94
Stream	10.02	23.46	13.44	134.13

*Fixed amount (no GC); **Average amount of allocated memory upon GC events.

Figure 3 shows the frequency of GCs along with the average heap sizes when GC events occur. As shown, GC frequencies are in the range of 6 to 12 GCs per second. The use of streams causes about two times more GCs than iterators and for-each statements.

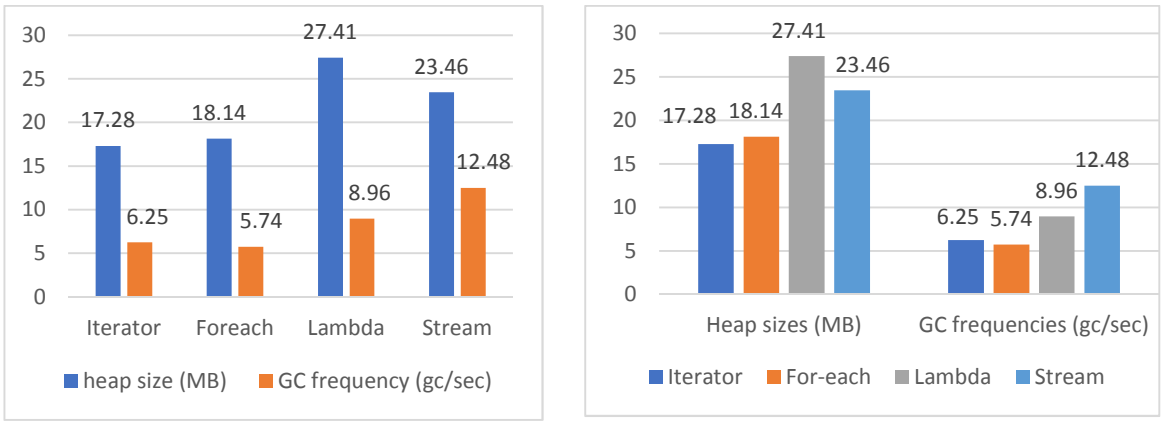

Figure 3 Heap sizes and GC frequencies

As described above, GC occurs more frequently for streams than other features. However, for a fair comparison we need to consider the *GC commencement threshold*, the low-memory threshold that triggers GC. GC thresholds are not fixed or the same. Figure 4 shows normalized GC frequencies of the Java features under investigation; they are estimated GC frequencies under the assumption that their average amounts of allocated memory upon GC events are the same as that of lambda code (27.41 MB). The stream code and lambda code need 2.8 and 2.4 times more GCs than the for-each code.

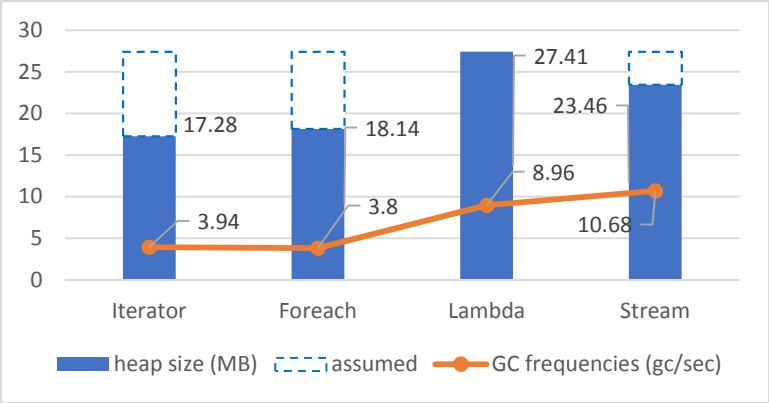

Figure 4. Normalized GC frequencies

One interesting finding is that, for a lambda expression, the way its body is written affects the memory performance of the expression. In particular, there is a noticeable difference in the amount of allocated memory depending on whether a lambda body refers to a field or not. A lambda body may refer to an (effectively final) local variable and a field. A final local variable can't be changed, however a field may be freely changed in the lambda body and the changes need to be propagated to the outside the lambda expression. This difference apparently affects the translation of lambda expressions such a way that there is noticeable difference in memory overheads. As shown in Table 4 below, the use of a field in the lambda body requires 4.31 MB (19 %) more memory and causes a 35% increase in GC frequency.

Table 4. Impact of a field reference appearing in a lambda body

Field ref.	# loops (/gc)	Time (msec)	Heap (MB)	GC freq (gc/sec)	Nor. GC freq (gc/sec)
Yes	2106384	111.57	27.41	8.96	8.96
No	2278409	126.86	23.10	7.88	6.64

A lambda expression is evaluated at runtime to an object -- e.g., an instance of *edu.utep.cs.java8demo.-$Lamda$6* that implements the functional interface of the lambda expression -- and its size is different depending on the existence of a field reference in its body; the lambda class itself is also created dynamically at runtime (see below for more on the translation of lambda expressions). For our sample lambda code, the size of a lambda object is 8 bytes when there is no field reference in the body. However, it increases to 16 bytes when the body contains field references; it's 16 bytes regardless the number of referenced fields in the body.

Why do the Java features under investigation cause the memory overheads? To find an answer, we tracked memory allocations of our experiment app using the Memory Monitor tool of Android Studio. The tool can track memory allocations for new objects along with their counts and total memory sizes. A sample screenshot of tracking is shown in Figure 5, and the tracking results for the Java features under investigation are summarized in Table 5.

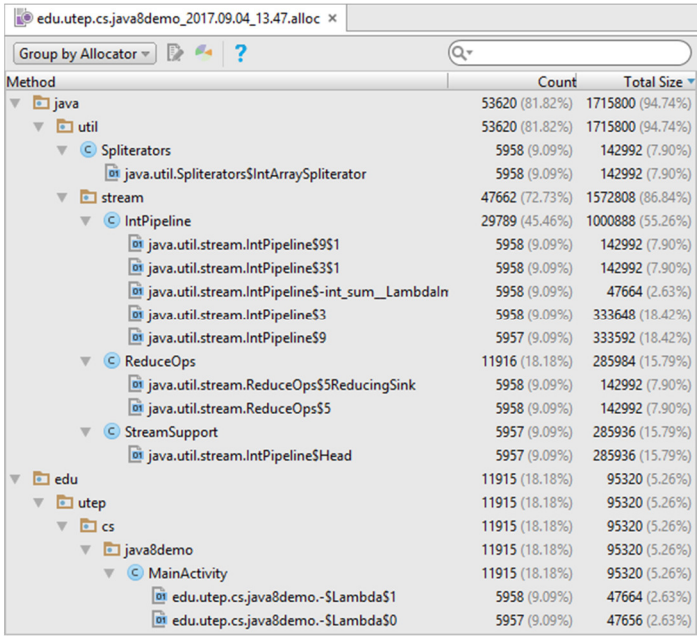

Figure 5. Memory allocation tracking

Table 5. Memory allocations

Feature	Allocator/Classes	# of objects		Total size (byte)	
		count	%	size	%
Iterator	java.util.AbstractList$Itr	65535	100.00	1572840	100.00
For-each	java.util.AbstractList$Itr	65531	99.99	1572696	99.99
	java.lang.ref.FinalizerReference	2	0.00	80	0.01
	MainActivity$GcWatcher	1	0.00	24	0.00
	java.lang.WeakReference	2	0.00	32	0.00
Lambda	MainActivity.-$Lambda$6	65535	100.00	1048560	100.00
Stream	java.util.Spliterators$IntArraySpliterator	5958	9.09	142992	7.90
	java.util.stream.IntPipeline$... (5 classes)	29789	45.46	1000888	55.26
	java.util.stream.ReduceOps$...(2 classes)	11916	18.18	285984	15.79
	java.util.stream.IntPipeline$Head	5957	9.09	285936	15.79
	MainActivity.-$Lambda$1	5958	9.09	47664	2.63
	MainActivity.-$Lambda$2	5957	9.09	47656	2.63

As shown in the table, nearly all memory allocations are due to hidden objects. For features like iterators, it is somewhat obvious because an object of the *Iterator* interface needs to be allocated and initialized. That

is, the call *values.iterator*() in our sample code creates an instance of *java.util.AbstractList$Itr*. It's similar for the for-each statement because it is at least conceptually translated to a loop that uses the iterator of the *Iterable* object; the *Iterable* interface defines the *iterator*() method that returns an iterator. For lambda expressions there are also hidden objects involved as described before. The evaluation of a lambda expression is similar to an evaluation of a class instance creation expression (see Section 15.27.4 of [8]). Either a new instance of a class (e.g., *edu.utep.cs.java8demo.-$Lambda$6*) that implements the target functional interface is allocated and initialized, or an existing instance is referenced. In Java, the implementation class is dynamically generated at runtime. However, the lambda body itself is typically translated into a static method of the class where the lambda expression appears. For streams, depending on the stream operations used, different types of hidden objects are allocated and initialized along with lambda objects for the arguments of the stream operations. As shown in Table 5, our sample stream code with three stream operations (*filter*, *map*, and *sum*) creates 11 different types of objects. The (average) size of a hidden object created (total size / # of objects) is: 24 bytes for iterators and for-each, 16 bytes for lambda, and 15 for streams.

As mentioned in the previous section, we also performed our experiment on Android virtual devices of different configurations. First, we varied both the RAM and the VM heap sizes of the devices: 512/16 MB (the minimum values allowed by Android Studio) vs. 1024/80 MB. As shown by the bar graph in Figure 6, there is no noticeable difference in the amounts of allocated memory as well as GC frequencies.[3] This is perhaps because the average GC thresholds (7-20 MB) are below the amount of the available memory.

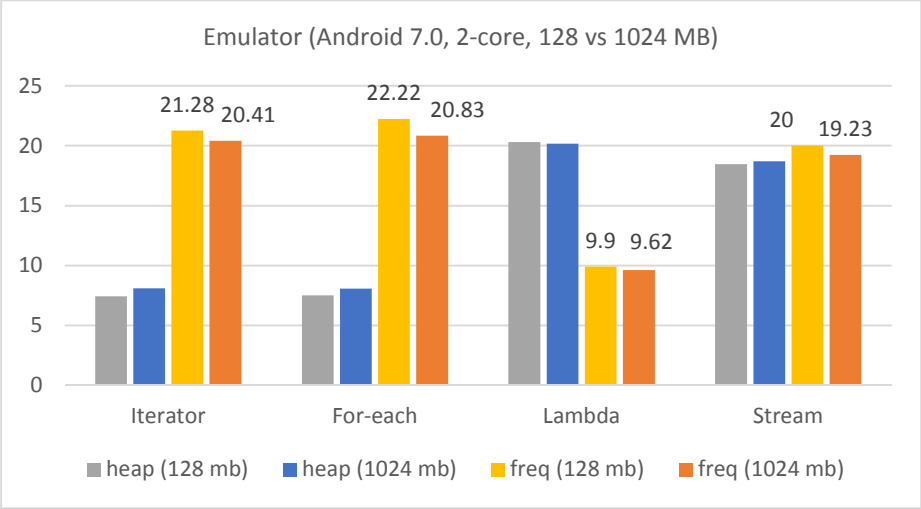

Figure 6. Emulators with different heap size, 128 MB and 1025 MB.

At the time of writing, Android Studio provides an experimental feature for specifying the number of processing units (cores) of the virtual devices. It can be either 1 or 2. We ran our experiment on two different emulators with 1 and 2 cores, respectively. The results are surprising for both the lambda code and the stream code. Their GC frequencies are decreased significantly on the 2-core emulator, 2.2 and 2.4 times for the lambda and the stream, respectively (see Figure 7). For the stream code, its average heap size on the 2-

[3] For lambda and streams, the amounts of allocated memory are frequently more than the specified VM heap size of 16 MB.

core emulator is about twice of the 1-core emulator, and this will definitely help in reducing the GC frequency on the 2-core (see Figure 8). However, the average heap size of the lambda code is almost the same on both the 1- and 2-core emulators. It looks like that, on a faster device with multiple cores, there are less memory overheads (GC frequencies) caused by the Java features under investigation. This is indeed the case when we compare the results of our experiment on the Samsung device with those of the emulators. The Samsung device (with 4 cores) outperforms the emulator (with 1 or 2 cores) up to 6 times in GC frequency. We do not know the correlation between GC frequency and the number of cores, although we know that ART garbage collectors run more efficiently than Dalvik virtual machine by taking advantage of multiple cores; GC runs on a core different from the those for the app code.

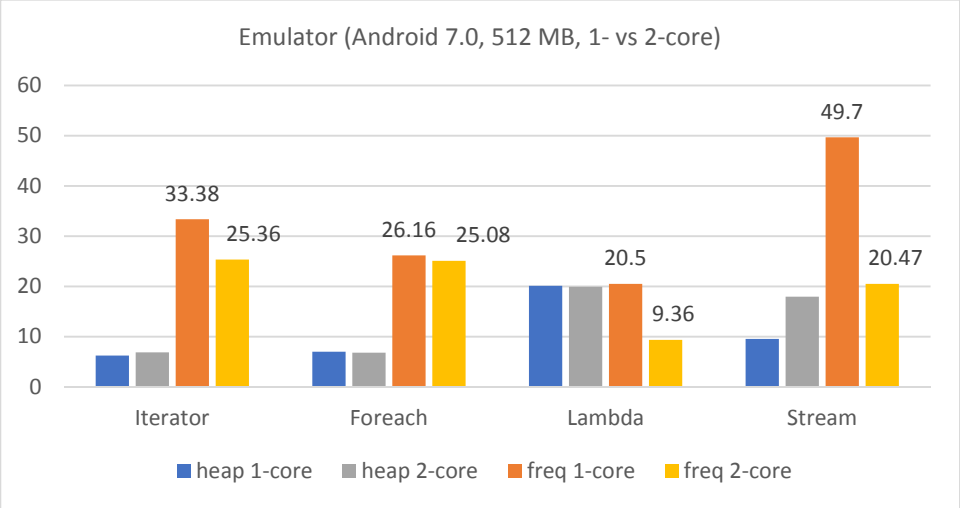

Figure 7. Emulators with 1 and 2 cores

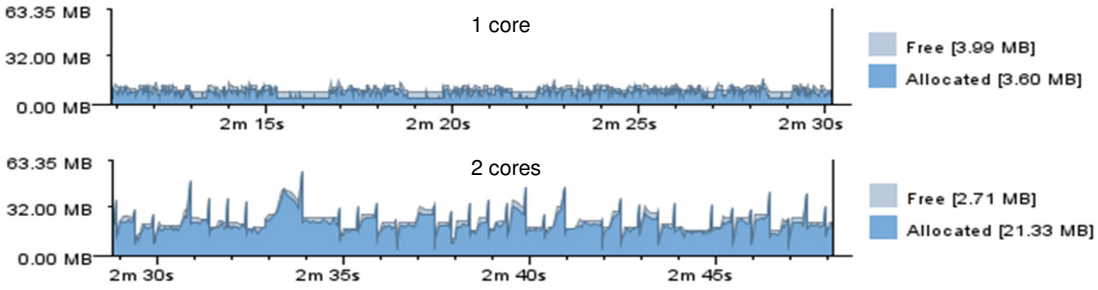

Figure 8. GC patterns for the stream code on 1 and 2-core emulators

6. Discussion

We mentioned in the previous section that our experiment is sort of a worst case analysis in that we run our experiment code repeatedly and continuously in an infinite loop. We can find out the exact amount of additional memory needed for each feature under investigation. We can use the Memory Monitor tool of

Android Studio to track memory allocations for new objects along with their counts and total memory sizes (see Table 5 in Section 5). If we know how many new objects are created per execution of the feature, we can calculate the overhead. We can also use our experiment code to count the number of loops in GC cycles. In Table 6 below, the second column (*Avg. # loops*) shows the average number of loops per GC cycle, and the last column shows the average amount of additional memory needed for the features under investigation. The table shows that streams have 34 times more overheads than lambda expressions. In our experiment, the stream code creates two lambda objects and many other hidden objects determined by stream operations (see Table 5). Even if our experiment is a worst case analysis, there are indeed such cases. A common example is code to be called by UI -- especially an animation UI -- directly or indirectly. The screen refresh rate of most Android device is 60 Hz, and thus the Android system may call the code up to 60 times per seconds. Another example is code handling streaming data; the code runs repeatedly, say once for each unit of the data stream [3]. If these features are used sparsely in an app, their impacts are insignificant as indicated by the numbers of iterations needed for GC events (the second column of the table). However, if used together in multiple places in an app, they might affect the memory performance of the app. It would be interesting future work to find out how frequently these language features are used in typical Android apps and what their collective impacts are on the overall memory performance of the apps.

Table 6. Memory overheads for features

Feature	Avg. # loops (/gc)	Allocated memory (MB)			Overhead (byte/loop)
		Without feature*	With feature**	Overhead	
Iterator	435113	6.54	17.28	10.74	26
For-each	408477	6.54	18.14	11.06	28
Lambda	2106384	6.67	27.41	20.74	10
Stream	41265	10.02	23.46	13.44	342

*Fixed heap size (no GC); **Average size of allocated heap memory when GC are initiated.

The memory inefficiencies of all the Java features investigated in our experiment are caused by hidden objects. One concern is that programmers may not aware of the existence of these hidden objects because they are all allocated and initialized by the compiler behind the scene. Another concern is that they may depend on how a compiler translates or evaluates at runtime these language features. The Java language specification, for example, doesn't constrain much on the way lambda expressions should be translated to or evaluated at runtime; a new (lambda) instance may be allocated and initialized, or an existing one may be reused as long as a few specified properties are satisfied (see Section 15.27.4 of [8]). A similar problem was reported in [2] in that iterators, for-each statements, choice of data structure and use of local variables affect the memory efficiency of Android apps. It is also said that frequent garbage collection may affect the battery power of a device by overheating it [3]. It is bothering that one has to pay close attention to hidden objects caused by not only the standard library classes and methods but also built-in language constructs like for-each statements and lambda expressions.

Identifying an app's performance bottlenecks and address them is critical to the success of the app [9]. For example, an Internet blog post says that more than 86% of users have uninstalled apps after using them only once due to poor performance [15]. As expected, there are performance tips, guidelines, patterns, code smells (anti-patterns), and best practices suggested by different people, some specifically for minimizing garbage collection execution time [2] [6] [9] [10] [12]. Our experiment reported in this document answers a more fundamental question -- memory overheads of Java language constructs and standard library classes. Our finding is unpleasant in that one also has to worry about the memory efficiency of Java language features in addition to their own code and the APIs used.

It would be interesting future work to find a way for specifying formally the memory requirements of Java language constructs, standard library classes, and other program modules. Ideally, such specifications should be part of the language and API specifications because they provide a formal basis for reasoning about the memory requirement and efficiency of an app.

References

[1] Joshua Bloch, *Effective Java*, second edition, Addison-Wesley, 2008.

[2] Yoonsik Cheon, *Are Java Programming Best Practices Also Best Practices for Android?* Technical Report 16-76, Department of Computer Science, University of Texas at El Paso, El Paso, TX, December 2016.

[3] Yoonsik Cheon, Rodrigo Romero, and Javier Garcia, HifoCap: An Android App for Wearable Health Devices, *Advances in Digital Technologies, Proceedings of the 8-th International Conference on Applications of Digital Information and Web Technologies*, Volume 295 of *Frontiers in Artificial Intelligence and Applications*, pages 178--192, IOS Press, 2017

[4] Eric Gamma, et al., *Design Patterns*, Addison-Wesley, 1994.

[5] Google Inc., *Android Monitor Overview*. Available from https://developer.android.com/studio/profile/android-monitor.html. Retrieved on 9/05/2017.

[6] Google Inc., *Best Practices for Performances*. Available from https://developer.android.com/training/best-performance.html. Retrieved on 9/05/2017.

[7] Google Inc., *Debugging ART Garbage Collection*. Available from https://source.android.com/devices/tech/dalvik/gc-debug. Retrieved on 9/05/2017.

[8] James Gosling, et al., *The Java Language Specification, Java SE 8 Edition*, February, 2015, Available from: https://docs.oracle.com/javase/specs/jls/se8/html/index.html. February, 2015.

[9] Mario Linares-Vasquez, et al., How Developers Detect And Fix Performance Bottlenecks in Android Apps, *IEEE International Conference on Software Maintenance and Evolution*, September 2015, pages 352-361.

[10] Umme Ayda Mannan, et al., Understanding Code Smells in Android Applications, *Proceedings of the International Conference on Mobile Software Engineering and Systems*, Austin, Texas, May 16-17, 2016, pages 225 – 234.

[11] Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.

[12] Doug Sillars, *High Performance Android Apps: Improve Ratings with Speed, Optimizations, and Testing*, O'Reilly, 2015.

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

[14] Karim Yaghmour, *Embedded Android: Porting, Extending, and Customizing*, O'Reilly, 2013.

[15] Junfeng Yang and Sarvar Dhillon, *Why You Should Care about Your Android App's Performance*, September 3, 2015. Available from http://blog.nimbledroid.com/2015/09/03/why-you-should-care-about-app-performance.html.