

## Cloud Security Anxiety?

Read This E-book to See 10 Risks Found in AWS Deployments & How to Remediate Them Palo Alto Networks

## Java 8 Streams - Lazy evaluation

[Updated: Jan 28, 2017, Created: Sep 27, 2016]

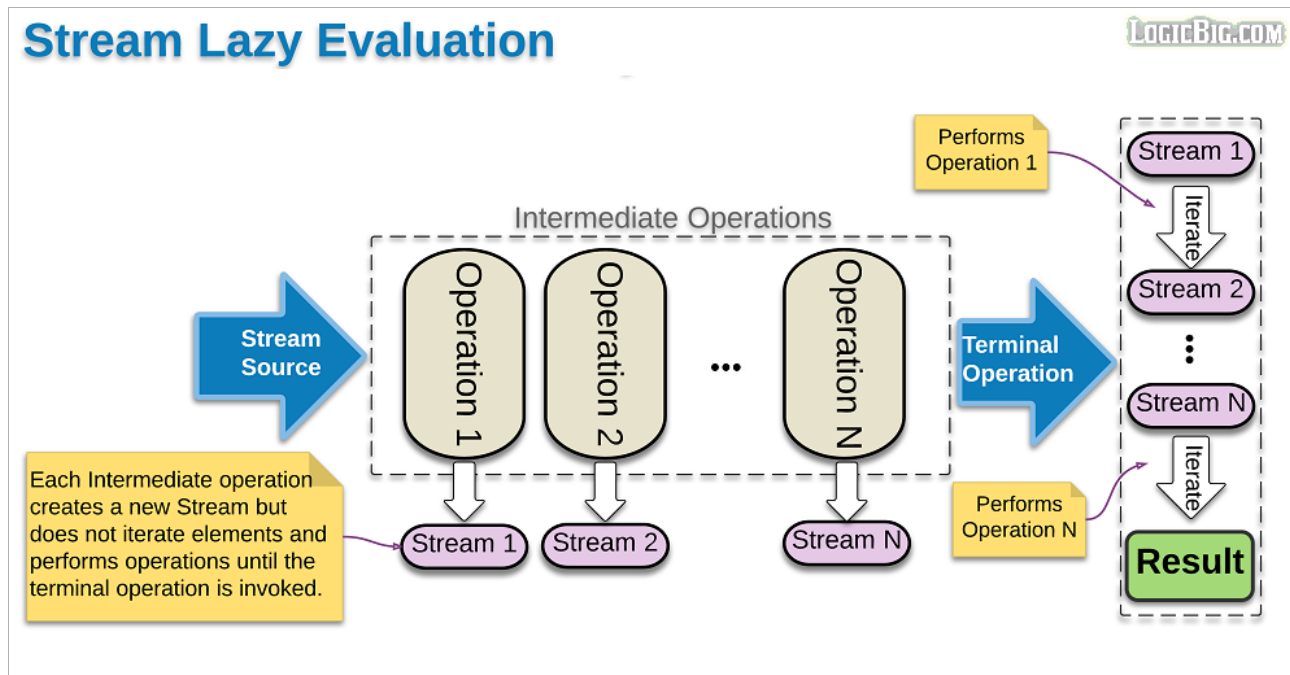
Streams are lazy because intermediate operations are not evaluated until terminal operation is invoked.

Each intermediate operation creates a new stream, stores the provided operation/function and return the new stream.

The pipeline accumulates these newly created streams.

The time when terminal operation is called, traversal of streams begins and the associated function is performed one by one.

**Parallel streams** don't evaluate streams 'one by one' (at terminal point). The operations are rather performed simultaneously, depending on the available cores.



## Lazy evaluation in sequential stream

```
public static void main (String[] args) {
    IntStream stream = IntStream.range(1, 5);
    stream = stream.peek(i -> log("starting", i))
        .filter(i -> { log("filtering", i);
                       return i % 2 == 0; })
        .peek(i -> log("post filtering", i));
    log("Invoking terminal method count.");
    log("The count is", stream.count());
}
```

```
public static void log (Object... objects) {
    String s = LocalDateTime.now().toString();
    for (Object object : objects) {
        s += " - " + object.toString();
    }
}
```

```

    }
    System.out.println(s);
    // putting a little delay so that we can see a clear difference
    // with parallel stream.
    try {
        Thread.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

Output:

```

19:04:55.062 - Invoking terminal method count.
19:04:55.072 - starting - 1
19:04:55.074 - filtering - 1
19:04:55.076 - starting - 2
19:04:55.077 - filtering - 2
19:04:55.078 - post filtering - 2
19:04:55.079 - starting - 3
19:04:55.080 - filtering - 3
19:04:55.081 - starting - 4
19:04:55.082 - filtering - 4
19:04:55.083 - post filtering - 4
19:04:55.084 - The count is - 2

```

Above output shows that all iteration and function evaluation begins only after invoking the terminal method `Stream#count()`.

In above example, we used the method `Stream#peek()`, note that this method is recommended for logging purposes only. We shouldn't perform stateful operations or apply side effects in this function. Here's the [API note](#) :

⚠ This method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline

## Lazy evaluation in parallel stream

```

public static void main (String[] args) {
    IntStream stream = IntStream.range(1, 5).parallel();
    stream = stream.peek(i -> log("starting", i))
        .filter(i -> {log("filtering", i);
            return i % 2 == 0;})
        .peek(i -> log("post filtering", i));
    log("Invoking terminal method count.");
    log("The count is", stream.count());
}

```

Output:

```

19:06:19.604 - Invoking terminal method count.
19:06:19.616 - starting - 3
19:06:19.616 - starting - 1
19:06:19.616 - starting - 4
19:06:19.616 - starting - 2
19:06:19.617 - filtering - 4
19:06:19.617 - filtering - 2
19:06:19.617 - filtering - 1
19:06:19.617 - filtering - 3
19:06:19.618 - post filtering - 2
19:06:19.618 - post filtering - 4
19:06:19.620 - The count is - 2

```



## What are the advantages of Laziness?

Lazy operations achieve efficiency. It is a way not to work on stale data. Lazy operations might be useful in the situations where input data is consumed gradually rather than having whole complete set of elements beforehand. For example consider the situations where an infinite stream has been created using `Stream#generate(Supplier<T>)` and the provided `Supplier` function is gradually receiving data from a remote server. In those kind of the situations server call will only be made at a terminal operation when it's needed.

Also consider a stream on which we have already applied a number of the intermediate operations but haven't applied the terminal operation yet: we can pass around such stream within the application without actually performing any operation on the underlying data, the terminal operation may be called at very different part of the application or at very late in time.

## Example project

Dependencies and Technologies Used:

- JDK 1.8
- Maven 3.0.4



```
stream-lazy-examples
  src
    main
      java
        com
          logicbig
            example
              LazyExample.java
              LazyParallelExample.java
              LogUtil.java
  pom.xml
```

```
package com.logicbig.example;

import java.util.stream.IntStream;

import static com.logicbig.example.LogUtil.log;

public class LazyExample {

    public static void main (String[] args) {
        IntStream stream = IntStream.range(1, 5);
        stream = stream.peek(i -> log("starting", i))
            .filter(i -> {
                log("filtering", i);
                return i % 2 == 0;
            })
            .peek(i -> log("post filtering", i));
        log("Invoking terminal method count.");
        log("The count is", stream.count());
    }
}
```

#### Project Structure

```
stream-lazy-examples
  src
    main
      java
        com
          logicbig
            example
              LazyExample.java
              LazyParallelExample.java
              LogUtil.java
  pom.xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.logicbig.example</groupId>
  <artifactId>stream-lazy-examples</artifactId>
  <version>1.0-SNAPSHOT</version>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.5.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
          <encoding>UTF-8</encoding>
        </configuration>
      </plugin>
    </plugins>
```

```
</build>
</project>
```

```
package com.logicbig.example;

import java.util.stream.IntStream;

import static com.logicbig.example.LogUtil.log;

public class LazyExample {

    public static void main (String[] args) {
        IntStream stream = IntStream.range(1, 5);
        stream = stream.peek(i -> log("starting", i))
            .filter(i -> {
                log("filtering", i);
                return i % 2 == 0;
            })
            .peek(i -> log("post filtering", i));
        log("Invoking terminal method count.");
        log("The count is", stream.count());
    }
}
```

```
package com.logicbig.example;

import java.util.stream.IntStream;

import static com.logicbig.example.LogUtil.log;

public class LazyParallelExample {

    public static void main (String[] args) {
        IntStream stream = IntStream.range(1, 5).parallel();
        stream = stream.peek(i -> log("starting", i))
            .filter(i -> {
                log("filtering", i);
                return i % 2 == 0;
            })
            .peek(i -> log("post filtering", i));
        log("Invoking terminal method count.");
        log("The count is", stream.count());
    }
}
```

```
package com.logicbig.example;



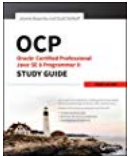





import java.time.LocalDateTime;

public class LogUtil {
    public static void log (Object... objects) {
        String s = LocalDateTime.now().toString();
        for (Object object : objects) {
            s += " - " + object.toString();
        }
        System.out.println(s);
        // just putting a little delay so that we can see a clear difference
        // with parallel stream
        try {
            Thread.sleep(1);
        }
    }
}
```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

 <p>Modern Java in Action: Lambdas, streams, functional and reactive progra...</p> <p><b>\$44.05</b> <del>\$64.00</del></p> <p>(4)</p>	 <p>Java 8 in Action: Lambdas, Streams, and functional-style programming</p> <p><b>\$83.69</b></p> <p>(84)</p>	 <p>OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide: Ex...</p> <p><b>\$37.53</b> <del>\$60.00</del></p> <p>(66)</p>	 <p>Effective Java</p> <p><b>\$43.86</b> <del>\$64.00</del></p> <p>(79)</p>
 <p>Java Pocket Guide: Instant Help for Java Programmers</p> <p><b>\$13.43</b> <del>\$40.00</del></p> <p>(6)</p>	 <p>OCP Java SE 8 Programmer II Exam Guide (Exam 1Z0-809)</p> <p><b>\$35.89</b> <del>\$60.00</del></p> <p>(13)</p>	 <p>Java 8 Lambdas: Pragmatic Functional Programming</p> <p><b>\$18.35</b></p> <p>(33)</p>	 <p>Oracle Certified Professional Programmer Exam 1Z0-809</p> <p><b>\$38.24</b> <del>\$44.00</del></p> <p>(16)</p>

## See Also

[Java 8 streams overview](#)

[Java 8 stream cheat sheet](#)

[Sequential vs parallel streams](#)

[java.util.stream](#)

[Short circuiting operations](#)

[Ordering](#)

[Stateful vs Stateless behavioral parameters](#)

[Side Effects](#)

[Reduction](#)

[Mutable Reduction](#)

[Java 8 Streams quick examples](#)

## Java 11 Tutorials

[Java 11 Features](#)

## Java 10 Tutorials

[Java 10 Features](#)

## Java 9 Tutorials

[Java 9 Module System](#)

[Java 9 Misc Features](#)

[Java 9 JShell](#)

## Recent Tutorials

[Spring Cloud - Hystrix CircuitBreaker, Thread Local Context Propagation](#)

[Spring Cloud - Circuit Breaker Hystrix Event Listener](#)

[Spring Cloud - Circuit Breaker Hystrix, Changing Default Thread Pool Properties](#)

[Spring Cloud - Circuit Breaker Hystrix, concurrent requests and default thread pool size](#)

[Spring Cloud - Circuit Breaker, Specifying Hystrix configuration in application.properties file](#)

[Spring Cloud - Hystrix Circuit Breaker, Setting Configuration Properties Using @HystrixProperty](#)

[Spring Cloud - Hystrix Circuit Breaker, getting failure exception in fallback method](#)

[Spring Cloud - Circuit Breaker Hystrix Basics](#)

[TypeScript - Standard JavaScript's built-in objects Support](#)

[JavaScript - Async Processing with JavaScript Promise](#)

[TypeScript - Applying Mixins](#)

[JPA Criteria API - Modifying and Reusing Query Objects](#)

[JPA Criteria API - Delete Operations](#)

[JPA Criteria API - Update Operations](#)

[JPA Criteria API - Case Expressions with CriteriaBuilder](#)

Share 

