Using Java Lambda Expressions Correctly & Efficiently

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University

Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand how lambda expressions provide a foundational functional programming feature in Modern Java
  - & know how to use them correctly & effectively

Using Java Lambda Expressions Correctly & Effectively

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope


```
int answer = 0;

new Thread(() -> {
            answer = 42;
            System.out.println("The answer is " + answer);
            });
```

See github.com/douglascraigschmidt/ModernJava/tree/main/FP/ex4

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope

If the body of the lambda is a block of statements or the lambda has no value & is not a single void method invocation, you must use curly brackets

```
int answer = 0;

new Thread(() -> {
                answer = 42;
                System.out.println("The answer is " + answer);
                });
```

See stackoverflow.com/a/11145970

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables

```
int answer = 42;

new Thread(() ->
            System.out.println("The answer is " + answer));
```

An "effectively final" variable in Java is a variable that is not declared as final, but its value never changes after it's initialized

See www.linkedin.com/pulse/java-8-effective-final-gaurhari-dass

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables

This lambda expression can access the value of "answer," which is an effectively final variable whose value never changes after it's initialized

```
int answer = 42;

new Thread(() ->
            System.out.println("The answer is " + answer));
```

See www.linkedin.com/pulse/java-8-effective-final-gaurhari-dass

Using Java Lambda Expressions Correctly & Effectively

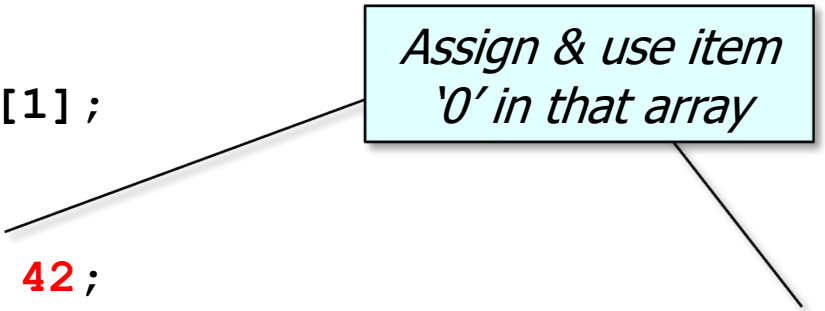
- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables
    - Here's one workaround for this restriction


```
int[] answer = new int[1];

new Thread(() -> {
            answer[0] = 42;
            System.out.println("The answer is " + answer[0]);
            });
...
return answer[0];
```

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables
    - Here's one workaround for this restriction

```
int[] answer = new int[1];

new Thread(() -> {
            answer[0] = 42;
            System.out.println("The answer is " + answer[0]);
            });
...
return answer[0];
```



Using Java Lambda Expressions Correctly & Effectively

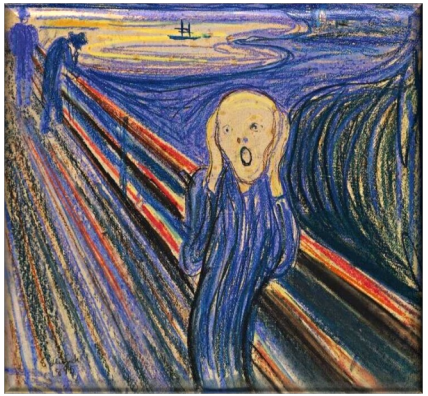
- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables
    - Here's one workaround for this restriction

```
int[] answer = new int[1];

new Thread(() -> {
            answer[0] = 42;
            System.out.println("The answer is " + answer[0]);
            });
...
return answer[0];
```

Do something with the updated array element after the lambda

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables
    - Here's one workaround for this restriction



```
int[] answer = new int[1];

new Thread(() -> {
            answer[0] = 42;
            System.out.println("The answer is " + answer[0]);
            });
...
return answer[0];
```

However, this solution incurs all the drawbacks of shared mutable state..

See henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables
    - Here's one workaround for this restriction
    - Here's another workaround

Create an atomic object

```
AtomicInteger answer = new AtomicInteger(0);

new Thread(() -> {
            answer.set(42);
            System.out.println("The answer is " + answer.get());
            });
...
return answer.get();
```

See www.digitalocean.com/community/tutorials/atomicinteger-java

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables
    - Here's one workaround for this restriction
    - Here's another workaround

```
AtomicInteger answer = new AtomicInteger(0);

new Thread(() -> {
            answer.set(42);
            System.out.println("The answer is " + answer.get());
            });
...
return answer.get();
```

Assign & use that atomic object

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables
    - Here's one workaround for this restriction
    - Here's another workaround

```
AtomicInteger answer = new AtomicInteger(0);

new Thread(() -> {
            answer.set(42);
            System.out.println("The answer is " + answer.get());
            });
...
return answer.get();
```

Do something with the updated atomic object after the lambda

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions cannot modify variables defined outside their scope
  - They can only access final or effectively final variables
    - Here's one workaround for this restriction
    - Here's another workaround


```
AtomicInteger answer = new AtomicInteger(0);

new Thread(() -> {
            answer.set(42);
            System.out.println("The answer is " + answer.get());
            });
...
return answer.get();
```

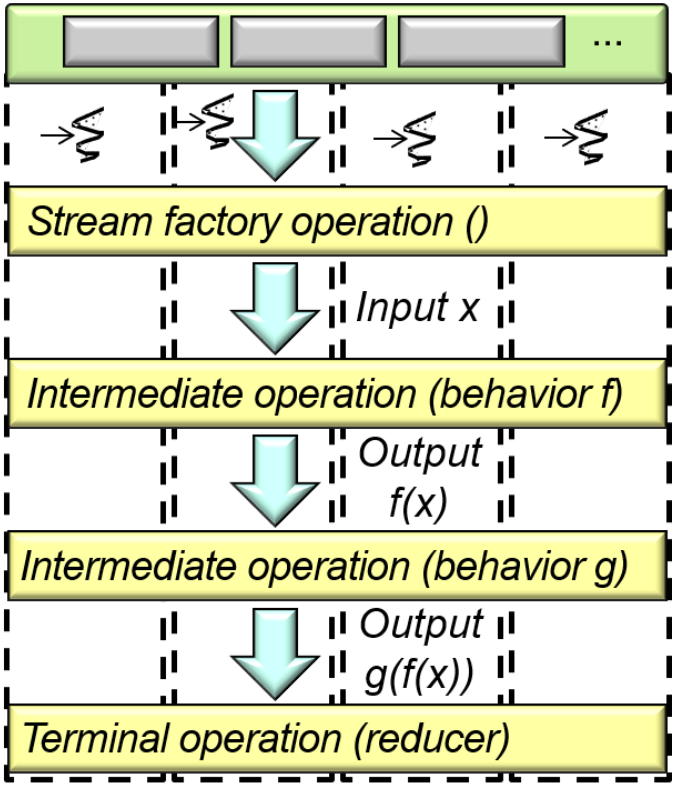
This solution is thread-safe, but incurs some (minor) synchronization overhead

See cephas.net/blog/2006/09/06/atomicinteger

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions are most effective when they are "stateless" & have no shared mutable state

Stateless lambda expressions are particularly useful when applied to Java parallel streams

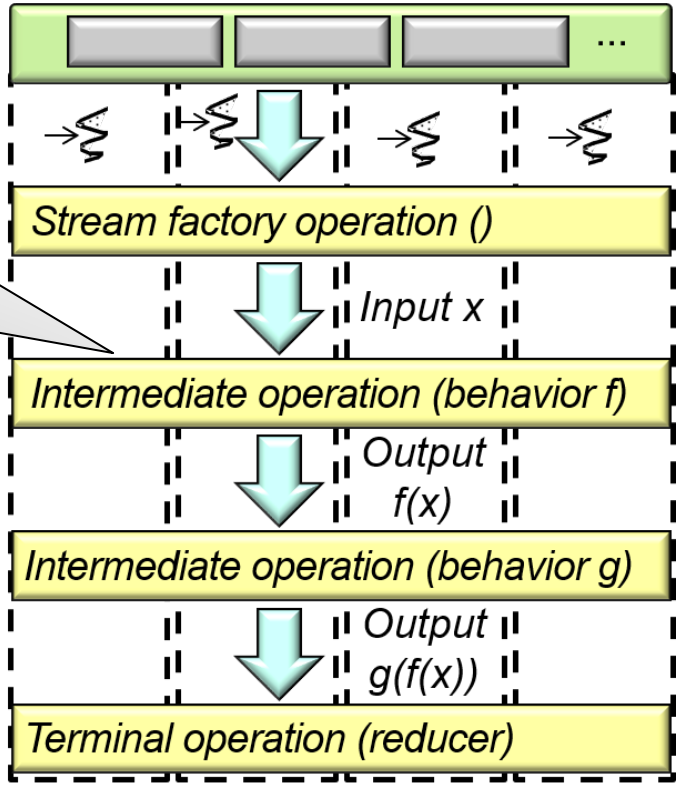


See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

Using Java Lambda Expressions Correctly & Effectively

- Lambda expressions are most effective when they are "stateless" & have no shared mutable state

```
String capitalize(String s) {
  if (s.length() == 0) return s;
  return s.substring(0, 1)
          .toUpperCase()
       + s.substring(1)
          .toLowerCase();
```



This 'pure function' is stateless

1. *Its return value is the same for the same arguments (no reading of shared mutable state)*
2. *Its evaluation has no side effects (no writing to shared mutable state)*

See en.wikipedia.org/wiki/Functional_programming#Pure_functions

End of Using Java Lambda Expressions Correctly & Efficiently