Applying Foundational Java 8 Features to a Concurrent Program

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 Lesson

- Understand how foundational Java 8 functional programming features are applied in the ThreadJoinTest program

See [github.com/douglascraigschmidt/LiveLessons/tree/master/ThreadJoinTest/updated](https://github.com/douglascraigschmidt/LiveLessons/tree/master/ThreadJoinTest/updated)
Learning Objectives in this Lesson

- Understand how foundational Java 8 functional programming features are applied in the ThreadJoinTest program
- Recognize the pros & cons of using Java 8 features in this example
Example of Starting & Joining Java Threads with Java 8
Example of Starting & Joining Java Threads with Java 8

- Use Java 8 features to start() & join() a group of threads to search for phrases in the works of William Shakespeare

```java
workerThreads
  .forEach(Thread::start);

workerThreads
  .forEach(thread ->
    { try { thread.join(); }
      catch (InterruptedException e)
        { ... }});
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/ThreadJoinTest/updated]
Example of Starting & Joining Java Threads with Java 8

• This program is "embarrassingly parallel"

See en.wikipedia.org/wiki/Embarrassingly_parallel
Example of Starting & Joining Java Threads with Java 8

- This program is "embarrassingly parallel"
- i.e., there are no data dependencies between worker threads

See en.wikipedia.org/wiki/Embarrassingly_parallel
There are several foundational Java 8 features to note.

Example of Starting & Joining Java Threads with Java 8
Example of Starting & Joining Java Threads with Java 8

- There are several foundational Java 8 features to note, e.g.,
  - Create/start worker threads via `forEach()` & a method reference

```java
public void run() {
    List<Thread> workerThreads = makeWorkerThreads
        (this::processInput);
    workerThreads
        .forEach(Thread::start);
...
}
```

`forEach()` & method reference
There are several foundational Java 8 features to note, e.g.,

- Create/start worker threads via `forEach()` & a method reference
- Pass a method reference to a method expecting a functional interface

Example of Starting & Joining Java Threads with Java 8

```java
public void run() {
    List<Thread> workerThreads =
        makeWorkerThreads
            (this::processInput);

    ...

    List<Thread> makeWorkerThreads
        (Function<String, Void> task) {
        ...
    }

    Void processInput(String input) {
        ...
    }
}
```

The use of a functional interface makes it easier to change that function is passed
There are several foundational Java 8 features to note, e.g.,

- Create/start worker threads via `forEach()` & a method reference
- Pass a method reference to a method expecting a functional interface
- Apply a function lambda to create the runnable processed by a thread

```java
List<Thread> makeWorkerThreads(Function<String, Void> task) {
    List<Thread> workerThreads = new ArrayList<>();
    mInputList.forEach(input ->
        workerThreads.add(new Thread(() ->
            new Thread(() -> task.apply(input))));
    return workerThreads;
}
```
Example of Starting & Joining Java Threads with Java 8

There are several foundational Java 8 features to note, e.g.,

- Create/start worker threads via `forEach()` & a method reference
- Pass a method reference to a method expecting a functional interface
- Apply a function lambda to create the runnable processed by a thread
- Wait for worker threads to finish

```java
public void run() {
    List<Thread> workerThreads =
        makeWorkerThreads
            (this::processInput);

    workerThreads
        .forEach(Thread::start);

    workerThreads
        .forEach(thread -> {
            ... thread.join(); ...
        } ...)
}
```

Uses `forEach()` & lambda expression
Example of Starting & Joining Java Threads with Java 8

- There are several foundational Java 8 features to note, e.g.,
  - Create/start worker threads via `forEach()` & a method reference
  - Pass a method reference to a method expecting a functional interface
  - Apply a function lambda to create the runnable processed by a thread
  - Wait for worker threads to finish

```java
public void run() {
    List<Thread> workerThreads =
        makeWorkerThreads
            (this::processInput);

    workerThreads
        .forEach(Thread::start);

    workerThreads
        .forEach(thread -> {
            ... thread.join(); ...
        }) ... 
```

*Simple form of barrier synchronization*

No other Java synchronization mechanisms are needed!
Pros of the ThreadJoinTest Program
Pros of the ThreadJoinTest Program

- Using foundational Java 8 features improves the program vis-à-vis original Java 7 version

See [github.com/douglascraigschmidt/LiveLessons/tree/master/ThreadJoinTest/original](https://github.com/douglascraigschmidt/LiveLessons/tree/master/ThreadJoinTest/original)
Pros of the ThreadJoinTest Program

• Using foundational Java 8 features improves the program vis-à-vis original Java 7 version, e.g.

• The Java 7 version has additional syntax & traditional for loops

```java
for (int i = 0; i < mInput.size(); ++i) {
    Thread t = new Thread (makeTask(i));
    mWorkerThreads.add(t);
}

... 
Runnable makeTask(int i) { 
    return new Runnable() {
        public void run() {
            String e = mInput.get(i);
            processInput(element);
        }
    }
}
... 
```
Pros of the ThreadJoinTest Program

- Using foundational Java 8 features improves the program vis-à-vis original Java 7 version, e.g.
- The Java 7 version has additional syntax & traditional for loops

```java
for (int i = 0; i < mInput.size(); ++i) {
    Thread t = new Thread (makeTask(i));
    mWorkerThreads.add(t);
}
...
Runnable makeTask(int i) {
    return new Runnable() {
        public void run() {
            String e = mInput.get(i);
            processInput(element);
        }
    }
    ...
```

The Java 7 version is thus more tedious & error-prone to program.
Pros of the ThreadJoinTest Program

• Using foundational Java 8 features improves the program vis-à-vis original Java 7 version, e.g.
  • The Java 7 version has additional syntax & traditional for loops
  • The Java 8 implementation is a bit more concise & extensible
  • Due to functional interfaces & basic declarative features

```java
public void run() {
    List<Thread> workerThreads =
        makeWorkerThreads
            (this::processInput);
    ...

    List<Thread> makeWorkerThreads
        (Function<String, Void> task) {
        ...

        mInputList.forEach(input ->
            workerThreads.add
                (new Thread(()
                    -> task.apply(input))));
```
Cons of the ThreadJoinTest Program
Cons of the ThreadJoinTest Program

- There’s still “accidental complexity” in the Java 8 version

Accidental complexities arise from limitations with techniques, tools, & methods
Cons of the ThreadJoinTest Program

• There’s still “accidental complexity” in the Java 8 version, e.g.
  • Manually creating/joining threads

```java
public void run() {
    List<Thread> workerThreads =
        makeWorkerThreads
            (this::processInput);

    workerThreads
        .forEach(Thread::start);

    workerThreads
        .forEach(thread -> {
            ... thread.join(); ...
        } ...)
```
Cons of the ThreadJoinTest Program

• There’s still “accidental complexity” in the Java 8 version, e.g.
  • Manually creating/joining threads
• Only one concurrency model supported
  • “thread-per-input” that hard-codes the # of threads to match the # of input strings

```
List<Thread> makeWorkerThreads(Function<String, Void> task){
    List<Thread> workerThreads = new ArrayList<>();
    mInputList.forEach(input ->
        workerThreads.add(new Thread(()
            -> task.apply(input)));
    return workerThreads;
}
```
Cons of the ThreadJoinTest Program

- There’s still “accidental complexity” in the Java 8 version, e.g.
  - Manually creating/joining threads
  - Only one concurrency model supported
- Not easily extensible without major changes to the code
  - e.g., insufficiently declarative
Cons of the ThreadJoinTest Program

• Solving these problems requires more than the foundational Java 8 features

Parallel Streams

- filter(not(this::urlCached))
- map(this::downloadImage)
- flatMap(this::applyFilters)
- collect(toList())

Completable Futures

- filter(not(this::urlCached))
- map(this::downloadImageAsync)
- flatMap(this::applyFiltersAsync)
- collect(toFuture())
End of Applying Foundational Java 8 Features