Applying Java 8 Functional Programming Features to a Parallel 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 Java 8 functional programming features are applied in a simple parallel 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 Java 8 functional programming features are applied in a simple parallel program
- This program searches for a list of phrases in the complete works of William Shakespeare

The Complete Works of William Shakespeare

Welcome to the Web's first edition of the Complete Works of William Shakespeare. This site has offered Shakespeare's plays and poetry to the Internet community since 1993.

See shakespeare.mit.edu
Learning Objectives in this Lesson

- Understand how Java 8 functional programming features are applied in a simple parallel program
- Recognize the pros & cons of using Java 8 features in this example

These “cons” help to motivate the need for Java 8 parallelism frameworks
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 (Exception e) {
    throw new RuntimeException(e);
  }
});
```

Starting SearchStream
- in thread 23 the phrase "Anon," was found at character offset 111628 in "The First Part of Henry VI"
- in thread 20 the phrase "Anon," was found at character offset 30949 in "The First Part of King Henry IV"
- in thread 20 the phrase "Anon," was found at character offset 48850 in "The First Part of King Henry IV"
- in thread 19 the phrase "Anon," was found at character offset 170485 in "The Tragedy of Hamlet"
- in thread 20 the phrase "Anon," was found at character offset 49402 in "The First Part of King Henry IV"
- in thread 20 the phrase "Anon," was found at character offset 49640 in "The First Part of King Henry IV"
- in thread 20 the phrase "Anon," was found at character offset 50003 in "The First Part of King Henry IV"
- in thread 20 the phrase "Anon," was found at character offset 50140 in "The First Part of King Henry IV"
- in thread 20 the phrase "Anon," was found at character offset 50464 in "The First Part of King Henry IV"
- in thread 20 the phrase "Anon," was found at character offset 50486 in "The First Part of King Henry IV"
- in thread 20 the phrase "Anon," was found at character offset 51628 in "The First Part of King Henry IV"
- in thread 20 the phrase "Anon," was found at character offset 52190 in "The First Part of King Henry IV"
- in thread 21 the phrase "Anon," was found at character offset 67832 in "Second Part of King Henry IV"
- in thread 16 the phrase "Anon," was found at character offset 34971 in "The Tragedy of Macbeth"
- in thread 40 the phrase "Anon," was found at character offset 37045 in "The Tragedy of Romeo & Juliet"
- in thread 40 the phrase "Anon," was found at character offset 46837 in "The Tragedy of Romeo & Juliet"

Ending SearchStream

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

- The program obtains the complete works of Shakespeare & a list of phrases from two text files

```java
List<String> mInputList =
    TestDataFactory.getInput(
        sSHAKESPEARE_DATA_FILE,
        "@");
```

...@The Tragedy of Hamlet
...

@The Tragedy of Julius Caesar
...

@The Tragedy of Macbeth
...

Each work begins with a ’@’ character
Example of Starting & Joining Java Threads with Java 8

- The program obtains the complete works of Shakespeare & a list of phrases from two text files

```java
List<String> mPhrasesToFind = TestDataFactory.getPhraseList(sPHRASE_LIST_FILE);
...
Neither a borrower nor a lender be
Beware the Ides of March
Brevity is the soul of wit
All that glisters is not gold
Sit you down, father; rest you
my kingdom for a horse!
...
```

Each phrase appears on a separate line
Example of Starting & Joining Java Threads with Java 8

Return the input data in the given file as an array of strings

```java
static List<String> getInput(String file, String splitter) {
    URI uri = ClassLoader.getSystemResource(file).toURI();

    String bytes = new String(Files.readAllBytes(Paths.get(uri)));

    return Pattern.compile(splitter).splitAsStream(bytes)
    .filter(((Predicate<String>) String::isEmpty).negate())
    .collect(toList());
}
```
Example of Starting & Joining Java Threads with Java 8

- Return the input data in the given file as an array of strings

```java
static List<String> getInput(String file, String splitter) {
    URI uri = ClassLoader.getSystemResource(file).toURI();

    String bytes = new String(Files.readAllBytes(Paths.get(uri)));

    return Pattern.compile(splitter).splitAsStream(bytes)
        .filter(((Predicate<String>) String::isEmpty).negate())
        .collect(toList());
}
```

Convert the file name into a path name.
Example of Starting & Joining Java Threads with Java 8

- Return the input data in the given file as an array of strings

```java
static List<String> getInput(String file, String splitter) {
    URI uri = ClassLoader.getSystemResource(file).toURI();

    String bytes = new String(Files.readAllBytes(Paths.get(uri)));

    return Pattern.compile(splitter).splitAsStream(bytes)
        .filter(((Predicate<String>) String::isEmpty).negate())
        .collect(toList());
}
```

Open the file & read all the bytes
Return the input data in the given file as an array of strings

```java
static List<String> getInput(String file, String splitter) {
    URI uri = ClassLoader.getSystemResource(file).toURI();

    String bytes = new String(Files.readAllBytes(Paths.get(uri)));

    return Pattern.compile(splitter).splitAsStream(bytes)
        .filter(((Predicate<String>) String::isEmpty).negate())
        .collect(toList());
}
```

Compile a regular expression used to split the file into a list of strings
Example of Starting & Joining Java Threads with Java 8

- Return the input data in the given file as an array of strings

```java
static List<String> getInput(String file, String splitter) {
    URI uri = ClassLoader.getSystemResource(file).toURI();

    String bytes = new String(Files.readAllBytes(Paths.get(uri)));

    return Pattern.compile(splitter).splitAsStream(bytes)
        .filter(((Predicate<String>) String::isEmpty).negate())
        .collect(toList());
}
```

*Filter out any empty strings in the stream*
Example of Starting & Joining Java Threads with Java 8

- Return the input data in the given file as an array of strings

```java
static List<String> getInput(String file, String splitter) {
    URI uri = ClassLoader.getSystemResource(file).toURI();

    String bytes = new String(Files.readAllBytes(Paths.get(uri)));

    return Pattern.compile(splitter).splitAsStream(bytes)
        .filter((Predicate<String>) String::isEmpty).negate()
        .collect(toList());
}
```

Collect the results into a list of strings
Return the phrase list in the file as a list of non-empty strings

```java
static List<String> getPhraseList(String file) {
    return Files.lines(Paths.get(ClassLoader.getSystemResource(file).toURI()))
        .filter(((Predicate<String>) String::isEmpty).negate())
        .collect(toList());
}
```
Example of Starting & Joining Java Threads with Java 8

- Return the phrase list in the file as a list of non-empty strings

```java
static List<String> getPhraseList(String file) {
    return Files
        .lines(Paths
            .get(ClassLoader.getSystemResource(file).toURI()))
        .filter(((Predicate<String>) String::isEmpty).negate())
        .collect(toList());
}
```

Read all lines from file into a stream
Example of Starting & Joining Java Threads with Java 8

- Return the phrase list in the file as a list of non-empty strings

```java
static List<String> getPhraseList(String file) {
    return Files
        .lines(Paths
            .get(ClassLoader.getSystemResource(file).toURI()))
        .filter(((Predicate<String>) String::isEmpty).negate())
        .collect(toList());
}
```

**Filter out any empty strings in the stream**
Example of Starting & Joining Java Threads with Java 8

• Return the phrase list in the file as a list of non-empty strings

```java
static List<String> getPhraseList(String file) {
    return Files
        .lines(Paths
            .get(ClassLoader.getSystemResource(file).toURI()))
            .filter(((Predicate<String>) String::isEmpty).negate())
        .collect(toList());
}
```

Collect the results into a list of strings
Example of Starting & Joining Java Threads with Java 8

- The main program creates & runs an instance of SearchOneShotThreadJoin

```java
public void main(String[] args) {
    new SearchOneShotThreadJoin().run();
}
```

Create/run an object to search for all phrases in parallel
Example of Starting & Joining Java Threads with Java 8

- Several key Java 8 features

```java
public void run() {
    // Start a group of threads that search for phrases in parallel
}```
Several key Java 8 features, e.g.

- Flexibly create worker threads via a factory method

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

Factory method makes a list of worker threads
Example of Starting & Joining Java Threads with Java 8

- Several key Java 8 features, e.g.
  - Flexibly create worker threads via a factory method
  - Pass a reference to a method expecting a functional interface

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

Void processInput(String input) {
    ... }

List<Thread> makeWorkerThreads
    (Function<String, Void> task)
    { ... }
```
Several key Java 8 features, e.g.
- Flexibly create worker threads via a factory method
- Pass a 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);
    ...
}

Void processInput(String input) {
    ...}

List<Thread> makeWorkerThreads(
    Function<String, Void> task)
    {
    ...}
```
Several key Java 8 features, e.g.

- Flexibly create worker threads via a factory method
- Pass a reference to a method expecting a functional interface
- Apply a function lambda to create runnable for a thread

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

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

This factory method creates a list of threads that will be joined when their processing is done.
Several key Java 8 features, e.g.

- Flexibly create worker threads via a factory method
- Pass a reference to a method expecting a functional interface
- Apply a function lambda to create runnable for a thread

Example of Starting & Joining Java Threads with Java 8

```java
List<Thread> makeWorkerThreads(Function<String, Void> task) {
    List<Thread> workerThreads = new ArrayList<>();

    mInputList.forEach(input ->
        workerThreads.add(new Thread(() ->
            task.apply(input))));

    return workerThreads;
}
```

Create an empty list of threads
Example of Starting & Joining Java Threads with Java 8

- Several key Java 8 features, e.g.
  - Flexibly create worker threads via a factory method
  - Pass a reference to a method expecting a functional interface
  - Apply a function lambda to create runnable for a thread

```java
List<Thread> makeWorkerThreads(
    Function<String, Void> task
) {
    List<Thread> workerThreads =
        new ArrayList<>();

    mInputList.forEach(input ->
        workerThreads.add
            (new Thread(()
                -> task.apply(input)));
    
    return workerThreads;
}
```

Create a thread for each input string to perform processing designated by the task parameter
Several key Java 8 features, e.g.
- Flexibly create worker threads via a factory method
- Pass a reference to a method expecting a functional interface
- Apply a function lambda to create runnable for a thread

Example of Starting & Joining Java Threads with Java 8

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

`task.apply()` creates a runnable that provides the computation for each of the threads
Example of Starting & Joining Java Threads with Java 8

- Several key Java 8 features, e.g.
  - Flexibly create worker threads via a factory method
  - Pass a reference to a method expecting a functional interface
  - Apply a function lambda to create runnable for a thread

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

*Add each new thread to the list*
Example of Starting & Joining Java Threads with Java 8

• Several key Java 8 features, e.g.
  • Flexibly create worker threads via a factory method
  • Pass a reference to a method expecting a functional interface
  • Apply a function lambda to create runnable for a thread

```java
List<Thread> makeWorkerThreads(Function<String, Void> task) {
    List<Thread> workerThreads =
        new ArrayList<>();

    mInputList.forEach(input ->
        workerThreads.add(
            new Thread(()
                -> task.apply(input)));

    return workerThreads;
}
```

Return the list of worker threads
Example of Starting & Joining Java Threads with Java 8

• Several key Java 8 features, e.g.
  • Flexibly create worker threads via a factory method
  • Pass a reference to a method expecting a functional interface
  • Start worker threads via `forEach()` & a method reference

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

See [docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html](http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html)

A Java thread is a unit of computation that runs in the context of a process.
Several key Java 8 features, e.g.

- Flexibly create worker threads via a factory method
- Pass a reference to a method expecting a functional interface
- 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
    start each worker thread to search for phrases in works of shakespeare
```
Several key Java 8 features, e.g.

- Flexibly create worker threads via a factory method
- Pass a reference to a method expecting a functional interface
- Start worker threads via `forEach()` & a method reference

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

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

This program uses a "thread-per-work" parallelism model
Several key Java 8 features, e.g.

- Flexibly create worker threads via a factory method
- Pass a reference to a method expecting a functional interface
- Start worker threads via forEach() & a method reference
- 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

- Several key Java 8 features, e.g.
  - Flexibly create worker threads via a factory method
  - Pass a reference to a method expecting a functional interface
  - Start worker threads via forEach() & a method reference
  - 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(); ...
        } ...)
```

See [en.wikipedia.org/wiki/Barrier_(computer_science)](en.wikipedia.org/wiki/Barrier_%28computer_science%29)
Example of Starting & Joining Java Threads with Java 8

- Several key Java 8 features, e.g.
  - Flexibly create worker threads via a factory method
  - Pass a reference to a method expecting a functional interface
  - Start worker threads via `forEach()` & a method reference
  - 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(); ...
        } ...)

No other Java synchronizers 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
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

List<Thread> makeWorkerThreads (Function<String, Void> task) {
  ...
  mInputList.forEach(input ->
    workerThreads.add
    (new Thread(() -> task.apply(input))));

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

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

  e.g., declarative Java 8 features such as forEach(), functional interfaces, method references, & lambda expressions
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 software techniques, tools, & methods

See [en.wikipedia.org/wiki/No_Silver_Bullet](en.wikipedia.org/wiki/No_Silver_Bullet)
Cons of the ThreadJoinTest Program

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

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

    workerThreads
        .forEach(Thread::start);

    workerThreads
        .forEach(thread -> {
            try { thread.join(); } 
            catch (Exception e) {
                throw new RuntimeException(e);
            }
        }); ... 
}
```

You must remember to start each thread!
Cons of the ThreadJoinTest Program

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

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

    workerThreads
        .forEach(Thread::start);

    workerThreads
        .forEach(thread -> {
            try {
                thread.join();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        });
...}
```

Note the verbosity of handling checked exceptions in Java 8 programs.

See [codingjunkie.net/functional-interface-exceptions](http://codingjunkie.net/functional-interface-exceptions)
Cons of the ThreadJoinTest Program

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

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

    workerThreads
        .forEach(Thread::start);

    workerThreads
        .forEach(ExceptionUtils
            .rethrowConsumer
            (Thread::join));
}
```

A helper class enables less verbosely use of checked exceptions in Java 8 programs

See stackoverflow.com/a/27644392/3312330
Cons of the ThreadJoinTest Program

- There’s still “accidental complexity” in the Java 8 version, e.g.
  - Manually creating, starting, & joining threads
- Only one parallelism model supported
  - “thread-per-work” hard-codes the # of threads to # of input strings

```java
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, starting, & joining threads
  • Only one parallelism 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

See www.dre.vanderbilt.edu/~schmidt/DigitalLearning
End of Applying Java 8 Functional Programming Features to a Parallel Program